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
Alexander Bigerl

Paderborn, September 15, 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. The demand for Semantic Web technologies in recent years motivated the construction of a variety of triple stores. Triple stores are a special kind of data management systems designed to store RDF triple data; a standard defined by World Wide Web Consortium (W3C) to represent the data web. Triple stores' performance is a crucial requirement to build scalable and reliable semantic web applications around major industrial fields, including medicine, commerce, and defense. For that, the scientific community dedicates a considerable amount of research to investigating approaches to designing high-quality RDF store solutions. Tentris is a novel triple store developed by the data science research group (DICE) at Paderborn university. It represents an RDF graph as a sparse boolean tensor and maps SPARQL operations to tensor operations. Tentris realizes the tensor concept using a trie-based data structure called Hypertrie that provides efficient translation of tensor operations. In my thesis, I further contribute to the Tentris project by investigating, implementing, and evaluating an approach to reduce Hypertrie memory utilization when storing large RDF data sets.

Contents

1 Introduction

RDF triple stores are considered central elements in the "Storage and Querying" phase of the linked data life cycle [5]. Tentris is a triple store that realizes the concept of storing the entire RDF graph in a sparse order-3 boolean tensor using a novel trie-based data structure, dubbed hypertrie. Among others, Tentris performance surpasses other well-known open-source and commercial triple stores [7] for single and multiple simultaneous SPARQL clients. Despite its high-performance characteristic, the Tentris system still suffers from excessive memory utilization when it tends to store large RDF graphs. hypertrie, on average, requires four times the space needed to store the RDF graph in a list. In my master thesis, I contribute to the development of Tentris by researching and implementing an approach to practically reduce the overall system memory footprint using a known technique in tree-based data structures called path compression. My effort is reflected mainly in extending the hypertrie code base to program the space reduction approach. The space reduction implementation is preserved in the GitHub branch: `https://github.com/dice-group/hypertrie/tree/path_compression`. Key contributions of the thesis are:

- Design an approach to reduce the utilized memory space in hypertrie.
- Expanding the hypertrie codebase to add an implementation to the approach using modern C++.
- Define and implement test cases (unit and integration tests) to check my implementation's conformance to the main hypertrie contracts defined in its interfaces.
- Evaluate the approach by executing bulk loading and benchmark experiments against real RDF data sets.

Outline I first introduce the main concepts upon which this work is built in chapter2. I present the idea of tagged pointers and provide a quick overview of the semantic web and introduce Tentris and the logic behind it in a bit of detail. Chapter 3 lists some contributions from the scientific community toward developing space-efficient in-memory data structures to store sequences. Then, the path compression approach to minimize the memory footprint of hypertrie is introduced in chapter 4. I discuss the approach's central concept and the refactoring work required on hypertrie on both the node design and the behavior levels. Testing the implementation of the space reduction approach against bulk loading experiments and benchmarks are shown in chapter 5. Chapter 6 concludes with a final discussion and future work.

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 the space-reduction solution. The first section 2.1 introduces common notations and terminologies used through out the work. Next, the pointer tagging technique is presented in section 2.2.

In the second part, the the semantic web topic is visited in section 2.3) before delving into into Tentris in section 2.4 where its concept and implementation are discussed.

2.1 Notation and Convention

For a function f, the domain and co-domain of f are denoted by $\mathrm{dom}(f)$ and $\mathrm{codom}(f)$, 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 $\{0, 1, ..., n-1\}$. Let $\mathbb{B}$ be the set of boolean values; i.e. $\mathbb{B} = \{true, false\}$. We map $true$ to 1 and $false$ to 0. 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 $\mathrm{dom}(t) = < 0, 1, ..., n-1 >$, is a tuple of t entries' positions.

2.2 Pointer Tagging

Pointer Tagging [16] is a low-level programming technique that uses the spare low bits in a pointer to encoding additional information. Using the Pointer tagging technique, pointer value (initially a memory address before tagging) can hold extra information, *a tag*. The tag can hold information about the point-to heap object or be used as meta-data to describe the pointer data usage further. Pointer tagging is mainly enabled because of how 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 how heap objects are arranged and accessed by the CPU. In modern computer architecture (say 64-bit architecture), CPUs

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 an ambiguous meaning. In the context of this work, we are targeting the generic term in the context of CPU architecture. A "processor word" refers to the size of a processor register or memory address register. The term word might refer to the size of a CPU instruction, or the size of a pointer depending on the exact CPU architecture [21]. 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 [21]. Let us take, as an example, the following coding snippet written in C++:

```
bool *p1 = new bool(true);     // p1 = 0x011F0
int  *p2 = new int(123);       // p2 = 0x11608
char *p3 = new char('A');      // p3 = 0x117A8
```

The snippet's execution will metaphorically result in a memory layout similar to the one presented in figure 2.1. According to C++ language specification [23], the size of an integer value in memory is 4 bytes, the size of char is 1 byte, and the short is 2 bytes. However, when we execute the previously mentioned statements, 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 to align them in memory. The same applies to the character value (Fig. 2.1).

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 data were to start somewhere in the middle of a word, the CPU would need to wait for 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 [21].

Many modern compilers implementations handle data alignment in memory automatically; Examples include C, C++, Rust, C# compilers.

2.2.2 Tagged Pointers

Some high-level programming languages, like C++, offer developers toolsets to work with memory. Using such toolsets, 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 a heap. Due to data alignment (Sec. 2.2.1), the memory address of any object in the heap memory is always $\alpha \cdot w$ where $w = 8$ (the word size). Hence, 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.

Consequently, all pointer values share that the first three low significant bits (LSBs) in their binary representation are always 0. As a result, we could exploit those bits to encode additional information without affecting the validity of the pointer (Fig. 2.2).

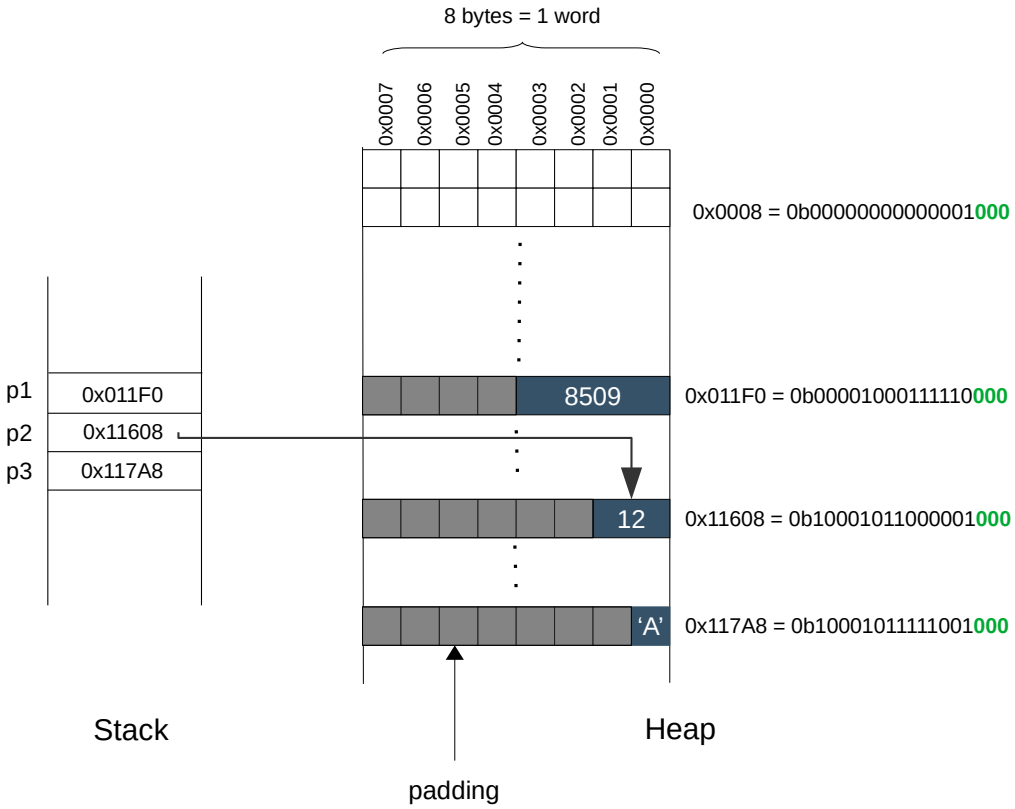


Figure 2.1: An example of memory layout with data objects of various types allocated in the heap. All objects are aligned by 8 bytes so their addresses are always multiple of 8.

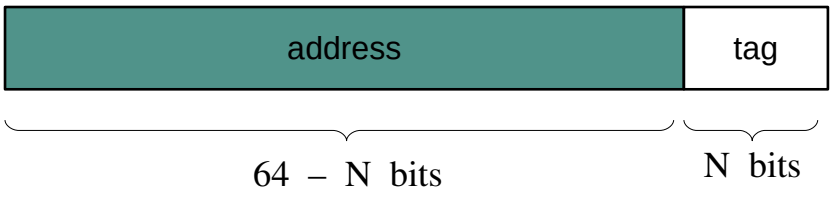


Figure 2.2: The structure of a tagged pointer. The number of tagging bits depends on the CPU architecture. Depending on the architecture, a $N = log_2(word\_size)$ of LSBs bits are dedicated for tagging. In 64-bit architecture, $word\_size = 8$, thus $N = 3$.

As a use case, depending on the tag, pointer tagging allows a dynamic representation of the pointer's numeric value. Hence, the actual payload of the pointer could represent 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 after a change in the tag value during run-time.

Tagged Pointer Implementation An approach to implement a tagged pointer is to develop a wrapper object around a pointer-typed variable. By that, we ensure safety by encapsulating the tagging logic inside the object scope and, as a result, presenting a single point of failure in case of pointer value misuse. The wrapper can be equipped with behaviors that govern the tag/ payload manipulation and retrieval. Accessing the pointer's binary representation to enable pointer tagging is straightforward in most programming languages and is done by using bit-wise operations.

2.3 Semantic Web

Today World Wide Web (WWW) holds a tremendous amount of data of different types(videos, images, text, geolocation data, books, publications, etc.). Such data comes from various sources (web applications, warehouse systems, GPS devices, smartphones, ATMs, etc.). However, such data are still processed passively by computer systems as there is no way to understand its meaning or context. Tim Berners-Lee coined the term *Semantic Web* for a Web of Data (or Data Web) [6] that can be processed by machines. The key technologies of Semantic Web are published by the World Wide Web Consortium (W3C). The Resource Description Framework (RDF), a standard for representing data in the Semantic Web, and SPARQL, a query language for RDF data, are introduced in this section.

2.3.1 Resource Description Framework

The Resource Description Framework (RDF) [1] is part of the W3C standard to define the web of data. Regardless of the data nature held on the web (blog posts, images, publications, newspaper articles, list of invoices, etc.), RDF identifies them uniformly as resources. In the standard, each resource is attached to a unique Internationalized Resource Identifier (IRI). IRI is a standard defined by the Internet Engineering Task Force in RFC 3987 [11]. Literals are another sort of resources. A literal comprises a hardcoded value represented as a string; ("Martin," "true", "12.3") are examples of literals. The third resource type is called a blank node. Blank nodes represent anonymous resources and always have local scope where they can be assigned a unique identifier. All definitions in this section are taken from "RDF 1.1 Concepts and Abstract Syntax" [26].

Definition 2.1 (RDF Terms)**.** Let I be the set of *IRIs*, L be the set of *literals* and be B the set of *blank nodes*. Further I, L and B are finite and pair-wise disjoint. Then the set $RT = I \cup L \cup B$ is called the set of *RDF terms*.

RDF Triple, RDF Graph

On an abstract level, an RDF triple can be seen as a statement that describes the relationship between two resources. An RDF triple can also serve to describe the property of a resource. RDF triple is represented by a sentence composed of three elements in order: *a subject*, *a predicate* and *an object*. The subject is an RDF resource. The subject has a property defined by the predicate and a value for that property set by the object. RDF restricts which RDF terms can be used for subject, predicate, and object:

Definition 2.2 (RDF Triple, RDF Graph)**.** An RDF Triple is a tuple $(s, p, o) \in$ where s is called subject, p is called predicate and o is called object.
A set of of RDF Triples is called RDF graph.

Example 2.1. An example of an RDF graph is given in Table 2.1. The data in the graph presents a list of pioneers, along with their occupations and spouses. Table 2.2 defines the abbreviations used in the example.

We use a particular query language to retrieve data from an RDF graph called SPARQL. SPARQL [15] is a recursive acronym for "SPARQL Protocol And RDF Query Language". Listing 2.1 shows a simple example of a SPARQL query against the RDF graph in table 2.1. The SPARQL language semantic is built on top of a formal language called SPARQL-algebra [15]. A core aspect of SRARQL-algebra is a sub-language called SPARQL-BGP (Basic Graph Pattern).

Definition 2.3 (Triple Pattern, Basic Graph Pattern)**.** Let V be an infinite set of variables disjoint to the set of RDF terms RT. The triple $(s, p, o) \in (I \cup B \cup V) \times (I \cup V) \times (RT \cup V)$ is called *triple pattern (TP)*. A set of triple patterns is called *basic graph pattern (BGP)*.

`<?person ex:occupation ex:Doctor>` is a triple pattern with a variable `?person` placed in the subject position. Applying a triple pattern tp to an RDF graph g (denoted by $P(g)$) finds all RDF triples where the predicate equals `ex:occupation` and object equals `ex:Doctor`. For each triple's occurrence, a mapping from `?person` to the corresponding subject of the triple is created. As a result, we end up with a set of mappings to variable `?person`. We call this set, the *solution mappings* denoted with Ω. Two solution mappings are said to be *compatible* when values assigned to the common variables are identical. Applying BGP on g is done by finding compatible solution mappings resulted from its triple patterns.

In SPARQL, raw `SELECT` operation is primarily based on evaluating basic graph patterns using the `join` operator defined in SPARQL algebra in the context of finding compatible solution mappings [15].

subject	**predicate**	**object**
ex:PresidentOfUS	rdf:type	madsrdf:Occupation
ex:Professor	rdf:type	madsrdf:Occupation
ex:PoliticalParty	rdf:type	v:Organization
ex:RepublicanParty	rdf:type	ex:PoliticalParty
ex:DemocraticParty	rdf:type	ex:PoliticalParty
ex:AndrewNg	madsrdf:occupation	ex:Professor
ex:BarakObama	madsrdf:occupation	ex:PresidentOfUS
ex:GeraldFord	madsrdf:occupation	ex:PresidentOfUS
ex:BarakObama	ex:party	ex:DemocraticParty
ex:GeraldFord	ex:party	ex:RepublicanParty
ex:BarakObama	foaf:spouse	ex:DemocraticParty
ex:GeraldFord	foaf:spouse	ex:MichelleObama
ex:AndrewNg	foaf:spouse	ex:CarolEReiley

Table 2.1: An example of an RDF graph.

abbr.	**IRI**
rdf:	http://www.w3.org/1999/02/22-rdf-syntax-ns
madsrdf:	http://www.loc.gov/mads/rdf/v1
ex:	https://www.example.com
v:	https://www.w3.org/TR/vcard-rdf/
foaf:	http://xmlns.com/foaf/0.1/

Table 2.2: The list of abbreviations used in table 2.1.

Listing 2.1: A SPARQL query that returns the spouse of each US president whose party is the democratic.

```
SELECT ?spouse
WHERE {
        ?president madsrdf:occupation ex:PresidentOfUS .
        ?president ex:party ex:DemocraticParty .
        ?president foaf:spouse ?spouse }
```

Triple Stores

Triple Stores are a special kind of data management system designed to store RDF triple data. It can store one or more RDF graphs. Generally, triple stores provide a standard interface to enable performing queries and other semantic operations on the stored RDF triples using a query language such as SPARQL (Sec. 2.3.1). Triple stores usually expose HTTP-based SPARQL endpoints for querying enablement. A triple store client can send a SPARQL query embedded in an HTTP request to the endpoint. The store evaluates the query against the RDF data it holds and returns the results to the client in the body of an HTTP response in a format conforms to the system configuration.

2.4 Tentris

There is no standard design guideline for triple stores. Hence different implementations of triple stores co-exist. Each subgroup of these implementations utilizes a category of underlying data structures and corresponding algorithms to govern the structures' behaviors. 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 triple stores' construction. The selection of internal data structures and the behavior definitions greatly influence the overall system efficiency.

Fuseki, Blazegraph, Virtuoso, and RDF-3X are popular implementations of Triple stores [25, 24, 13, 19]. One of the key design characteristics those triple stores have in common is that they all utilize B+ trees to store triples' representations. Other categories of triple stores use 3D Boolean tensors to store and process RDF data. Examples of such system include TensorRDF [10] and BitMat [4]. 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. In the following, a novel tensor-based RDF triple store is presented.

Tentris is a triple store variant designed by the data science research group at Paderborn university[7]. Tentris is an in-memory storage solution that represents RDF knowledge graphs as sparse order-3 tensors using a novel data structure, called Hypertire. It then uses tensor algebra to carry out SPARQL queries by mapping SPARQL operations to tensor operations, namely slicing and Einstein summation[12]. At the time of this writing, Tentris' SPARQL engine realizes *SPARQL-BGB* (a subset of the SPARQL-algebra). Consequently, the engine can execute SPARQL queries containing the following keywords: `SELECT`, `WHERE`, and `DISTINCT`[15].

Since this work represents a further contribution to the project Tentris, this section is dedicated to delivering a preface for the Tentris system design aspects. In sub-section 2.4.1, I specify what tensors are. A way of representing RDF graphs in tensors is showed in sub-section 2.4.2. Hypertrie data structure is visited in sub-section 2.4.3.

2.4.1 Tensor Algebra

In mathematics, the term *tensor* holds a representation-independent meaning. According to [20], tensors are "objects with many indices that transform in a specific way under a change of Basis". This mathematical construct has many applications in physics, artificial intelligence, and other fields. However, to apply tensors in a practical context, a more concrete definition should be selected. Among other choices, a multi-dimensional array is widely adopted as a representation of a tensor. Throughout this work, a finite n-dimensional array for tensor representation is considered. In the context of the Tentris project [7], the term tensor has the following formal definition:

Definition 2.4 (Tensor)**.** A mapping

$$T : \mathbf{K} \to V$$

from a multi-index $\mathbf{K}$ to a codomain V is called tensor. $\mathbf{K}$ is called *key basis* of dimension $n \in \mathbb{N}$ with

$$\mathbf{K} = K_0 \times K_1 \times ... \times K_{n-1}, K_i \subset \mathbb{N}$$

The tuple $\mathbf{k} \in \mathbf{K}$ is called *key*, K_i is called a *key part basis* and $k \in K_i$ is called *key part*.
The dimension of the key basis is also called dimension of the tensor and is denoted by $\text{ndim}(T)=n$.
Further, $\text{nnz}(T) = |\{\mathbf{k} \in \mathbf{K} | T(\mathbf{k}) \neq 0\}|$ denotes the *number of non-zero entries* in tensor T.

To resolve a value of a tensor, we use the array subscript notation $T[k_0, .., k_{n-1}]$. Moreover, the symbol $V^{\mathbf{K}}$ denotes the set of all mappings $T : \mathbf{K} \to V$. My work exclusively consider tensors T with $\mathbb{B}$ or $\mathbb{N}$ as codomain and only use multi-indexes with equal key part basis $K_1 = ... = K_n \subset \mathbb{N}$. An n-dimensional tensor is also called *order-n* tensor.

Example 2.2. In the following, some examples to illustrate the Definition 2.3:

1. A tensor $S \in \mathbb{Z}^{\emptyset}$ is called a *scalar*.
$$S = 1$$
So $S[\emptyset] = S[]$ is 1.

2. A tensor $X \in \mathbb{Z}^{\mathbb{N}_3}$ is called a *vector*.
$$X = \begin{bmatrix} 4 \\ 7 \\ 15 \end{bmatrix}$$
where $X[2]$ is 15.

3. Now, we take a three dimensional tensor $Y \in \mathbb{Z}^{\mathbb{N}_2 \times \mathbb{N}_2 \times \mathbb{N}_2}$. It can be visualized by a vector for the first key part which, in turn, has matrices each with dimensions corresponding to second and third key parts. As an example:
$$Y = \begin{bmatrix} \begin{bmatrix} 1 & 2 \\ 3 & 5 \end{bmatrix} \\ \begin{bmatrix} 7 & 11 \\ 13 & 15 \end{bmatrix} \end{bmatrix}$$
Such that, $Y[1, 0, 1]$ is 11.

Tensor Operations

This section highlights the core operations on tensors. In particular, I describe what tensor slicing and Einstein summations are. In the following, formal definitions of the operations are skipped. Instead, the section presents them informally with examples.

Slices Slicing is a useful operation to retrieve a well-defined portion of a tensor $T \in V^K$ in the form of a lower order tensor. Slicing is done by using a *slice key* $\mathbf{s} \in S = (K_0 \cup \{:\}) \times .... \times (K_{n-1} \cup \{:\})$, and denoted like as we are retrieving a value but with one or more dimensions not bound, e.g. $T[:, x, :]$ (or $T[<:, x, :>]$). When applying slice key $\mathbf{s}$ to a tensor T, the unbounded dimensions in the slice key (the ones that are marked with :) are kept. A slice key part $s_i \neq$: removes all entries with other key parts at position i and removes K_i from the resultant tensor domain. e.g., $T[:, 2, :]$ or $T[<:, 2, :>]$.

Example 2.3. Back to the third item in Example 2.2, slicing tensor Y by slice keys $s_1 =<:, 1, :>$ results in an order-2 tensor $Z_1 = Y[:, 1, :]$, and with a slice key $s_2 =< 0, 1, :>$ results in an order-1 tensor $Z_2 = Y[0, 1, :]$:
$$Z_1 = \begin{bmatrix} 3 & 5 \\ 13 & 15 \end{bmatrix}$$
$$Z_2 = \begin{bmatrix} 3 \\ 5 \end{bmatrix}$$

Worth to mention that, slicing operation is associative, meaning that applying a sequence of slicing operations with different grouping to tensor Y brings the same results:
$$Y[1, 0, :] = (Y[1, :, :])[0, :] = Y([:, 0, :])[1, :] = Z_2$$

Einstein Summation Einstein summation is a mighty operator that was first coined by Albert Einstein and presented to the scientific community in [12]. The behavior of Einstein summation is defined basically by assigning each tensor (operand) invovled an ordered list of key part basis labels. Each label in the list maps to a key part basis in a tensor (operand) involved in the summation. For example, a tensor $T \in V^{N_3,N_5,N_{10}}$ can have the following label list, T_{ijk} then i corresponds to N_3, j to N_5, and k to N_{10}. Consider the tensors Z_1 and Z_2 from the example 2.3, we perform an Einstein summation over Z_1 and Z_2 and store the results in R such that $R_i \leftarrow Z_{1_{ij}} \times Z_{2_j}$. As labels i is fixed in the result, R entries are evaluated like the following:

$$R[i \in N_2] = \sum_{j \in N_2} Z_1[i,j].Z_2[j]$$

.

Consequently, $R = \begin{bmatrix} 34 \\ 114 \end{bmatrix}$.

2.4.2 Storing RDF Graphs in Tensors

Let g be an RDF graph with IRIs (I), blank nodes (B) and literals (L) being finite sets, and thus the set of RDF terms RT is also finite (Sec. 2.3.1). We define a bijective function $id_{RT} : RT \to N_n$ where $|RT| = n$. id_{RT} maps each RDF term to a unique identifier. We call this function *index of resources*. Similarly, we define the *resources by indices* function as $res_{RT} = id_{RT}^{-1}$.

Every triple $< s, p, o > \in g$ is also in $(RT \times RT \times RT)$. For that, we can define an order-3 tensor $T \in \mathbb{B}^{N_n \times N_n \times N_n}$ with equal key part basis N_n. We store the triple $< s, p, o >$ in T by setting $T[id(s), id(p), id(o)] = 1$ and all other entries in T are set to 0. We call T the tensor representation of the RDF graph g and denoted by T_g. In other words:

$$T[i_s, i_p, i_o] = 1 \iff < res(i_s), res(i_p), res(i_o) > \in g$$

Since T_g has the codomain $\mathbb{B}$, we describe T_g as a *boolean tensor*. Subject, predicate, and object are associated with one dimension each.

Remark 2.1. We omit the use of the subscript RT with the functions' names id_{RT} and res_{RT} when it is clear from the context.

Remark 2.2. In practice, a dimension range ($|RT| = n$) of an RDF tensor T_g fits much more than the bounded resources' IDs in g for that dimension. In [7], however, it is proven that by choosing equal key part basis for enable efficient dimension matching using Einstein notation.

Remark 2.3. As T_g can store a super set of RDF triple in g, we call T_g *a sparse tensor*

Example 2.4. Figure 2.3 shows a 3D visualization of 3-dimensional tensor that represents the RDF graph g depicted in table 2.3. We notice that the number of distinct RDF terms in g is 7, so we can represent g with a tensor $T_g \in \mathbb{B}^{N_7 \times N_7 \times N_7}$.

Mapping SPARQL to tensors

The ability to bridge SPARQL algebra to tensor operations is a strict requirement to realize a tensor-based triple store. For instance, a translation of triple pattern and basic graph pattern, core building blocks of SPARQL semantic, to tensor operations enables the triple store to evaluate queries based on those patterns, such as `SELECT` queries.

subject	predicate	object
:e1 (1)	foaf:knows (2)	:e2 (3)
:e1 (1)	foaf:knows (2)	:e3 (4)
:e2 (3)	foaf:knows (2)	:e3 (4)
:e2 (3)	foaf:knows (2)	:e4 (5)
:e3 (4)	foaf:knows (2)	:e2 (3)
:e3 (4)	foaf:knows (2)	:e4 (5)
:e2 (3)	rdf:type (6)	dbr:Unicorn (7)
:e4 (5)	rdf:type (6)	dbr:Unicorn (7)

Table 2.3: A list of RDF triples representing an RDF graph g. Resources are printed alogn with their corresponding IDs (enclosed in brackets).

.

Triple Pattern A slice of a tensor does basically what triple pattern does to the corresponding RDF graph. Thus, a triple pattern tuple tp can be viewed as a slice key s in the tensor domain. Slice positions (**:**) in s are set at locations corresponding to the where variables are defined in tp. Other locations in s are set to concrete key parts (IDs) that maps to certain RDF terms defined in the tuple at the corresponding positions. Each variable in the triple pattern is associated with one dimension of the tensor. Now, evaluating a triple pattern tp against an RDF graph g, is equivalent to slicing the tensor T_g using slice key s. Then, the set of keys to non-zero elements in $S = T_g[s]$ are selected and the bounding key parts are assigned to the set of variables in tp at the designated positions.

Example 2.5. let tp be a triple pattern defined as `<?person ex:type ?type>` to be evaluated against the RDF graph g in table 2.1. An equivalent slice key s for tp is $s =$`<:, 6, :>`. Slicing the RDF tensor T_g (Fig. 2.3) using s results in an order-2 tensor with two non-zero entries (3, 7), (5, 7). Thus, the set of solution mappings is:
$\Omega = \{[?person \rightarrow res_{RT}(3), ?type \rightarrow res_{RT}(7)], [?person \rightarrow res_{RT}(5), ?type \rightarrow res_{RT}(7)]\}$.

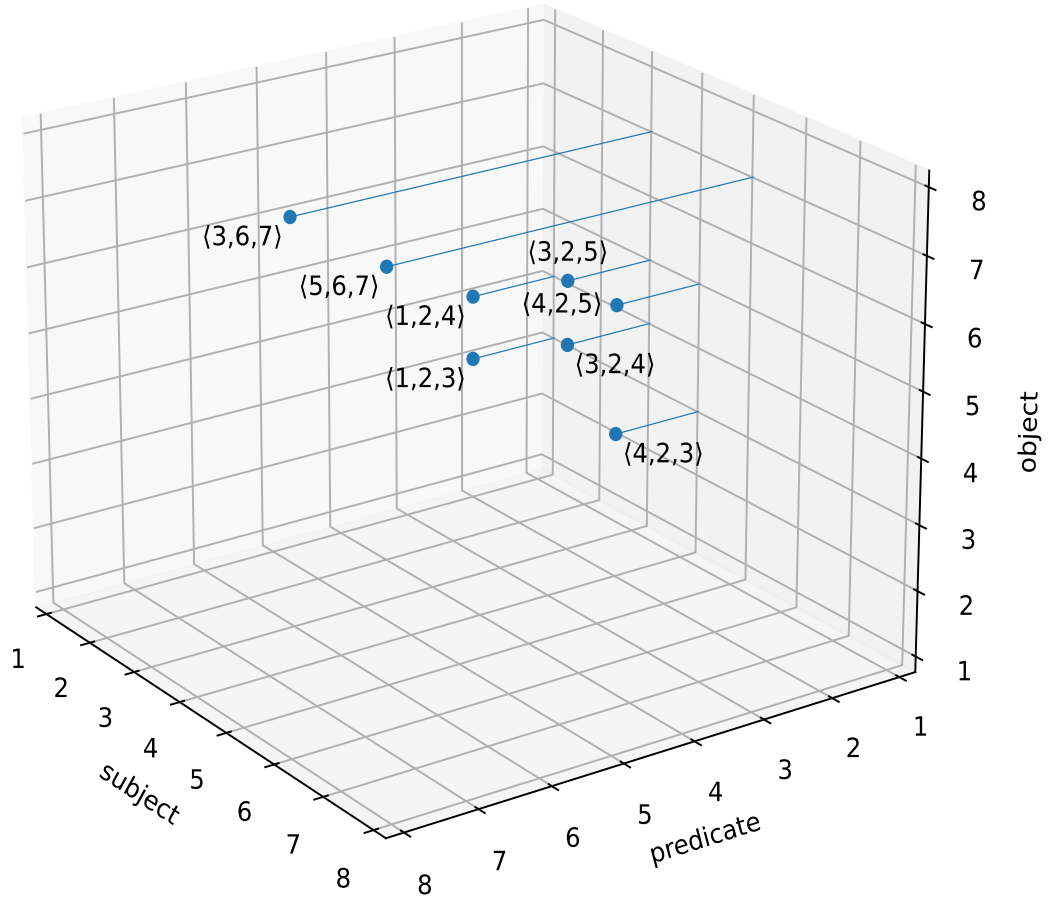


Figure 2.3: A 3D plot of an order-3 RDF tensor representing the RDF graph in table 2.3.

2.4.3 Tentris and Hypertrie

Underlying the order-d RDF tensor concept, Tentris realizes the concept using a novel data structure called Hypertrie. Before I discuss hypertrie structure, I first clarify how tensors storing RDF graphs can be represented as tries. Then, I show why this is not engouh for our use case. rephrase the concept.

Trie

A *Trie* [8] is a tree structure used to store *sequences of characters* from an alphabet $\mathbb{A}$. An example of an alphabet is the ASCII codes set. Sequences, in this case, are strings. A node in a Trie contains potentially one outgoing edge for each possible character. Each node in the tree corresponds to a prefix of some sequences of the set, so if the same prefix occurs several times, there is only one node to represent it.

A possible realization of a Trie node is to use a pointer array of the size $|\mathbb{A}|$ [8]. Each pointer points to either another trie node or null. Each array entry corresponds to a character in the alphabet. As an array lookup operation is computationally constant, looking up sequences in the Trie is fast and requires only $O(k)$ time where k is the sequence size. This approach, however, becomes space-inefficient as the size of the alphabet set increases or when it is infinite. An alternative way to store edges is to maintain a hash table HT whose size will increase as we add distinct edges. The hash table keys corresponds to *edge labels* (characters) and are mapped to the node pointers (edges). Following that, the structure can still be used efficiently to retrieve sequences as accessing keys in a well-implemented sparse hash table is nearly constant.

A *fixed-depth trie* C is a trie that holds sequences of the same length n, termed as *keys*. In that case, a sequence $l =< l_0, \ldots, l_m > \in \mathbb{A}^m$ of length $m <= n$ forms a *key prefix*. $C[l]$ is defined as the node that is reached from the root node r by walking along the nodes with edge labels equal to the entries of l. Hence, $C[l]$ could be undefined if no appropriate path exists. Moreover,

We can represent a tensor $T_g \in \mathbb{B}^{\mathbb{N}_n \times \mathbb{N}_n \times \mathbb{N}_n}$ that is used to store an RDF graph g with $|RT| = n$ (Sec. 2.4.2) using a fixed-depth trie C_{T_g} of depth $d = \text{ndim}(T_g) = 3$ with the alphabet $\mathbb{N}_n$. We call C_{T_g} a *a trie tensor or trie representation of* T_g, if:

$$\forall \mathbf{k} \in \mathbb{N}_n \times \mathbb{N}_n \times \mathbb{N}_n, v \neq 0 : T_g[k] = v \iff C_{T_g}[k] = v$$

Resolving a tensor trie C_T of depth d by key prefix $l =< k_0, ..., k_m >$ where $m < d$ is equavalent to slicing the corresponding d-order tensor using a slice key $s =< k_0, ..., k_m, e_{m+1}, ..., e_d >$ with $e_i = \mathbf{:}$. Then the resultant tensor slice $T[k_0, ..., k_m, e_{m+1}, ..., e_d]$ ($d^{'}$-order where $d^{'} = d - m$) is represented by an inner trie node or *subtrie* of depth $d^{'}$.

In the context of this writing, the term *node depth* corresponds to the sequences length that a node holds. For a trie tensor C_T of depth d. The root node is assigned the depth d, an inner node holding prefix of size m, has the depth $d^{'} = d - m$. Leaf nodes share the depth 1. An example of a trie tensor is showed in figure 2.4.

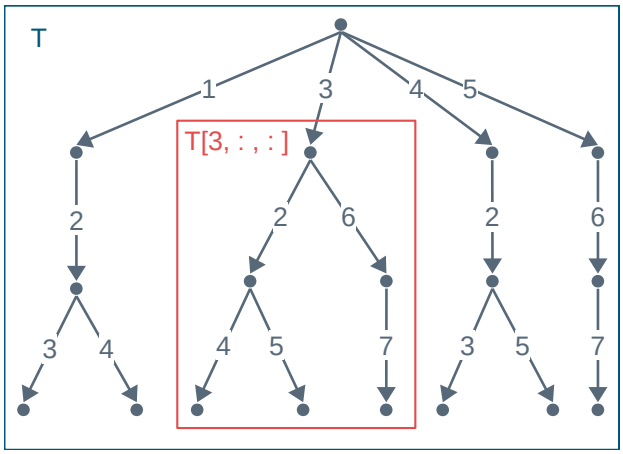


Figure 2.4: Trie representation of the tensor T_g that depicts the RDF graph g in Table 2.3. A slice $T_g[3, :, :]$ by the first dimension with 3 is shown in the red (inner) box.

Hypertrie

In the previous section we defined a trie as a data strucuture for realizing an order-3 RDF tensor. The structure is memory-efficient as it sparsely encodes a Boolean-valued tensor by storing only the keys that map to 1. As keys with the same key prefixes share the same node, a moderate level of compression is achieved. Slicing is also efficient if the set of edges to children are represented using a well-designed hashtable or search tree[**?**].

A trie, as defined earlier, however, still lake the ability for flexible slicing, that is by any combination of dimensions. The trie tensor as it is defined so far allows slicing positions to only suffix the slice keys, thus using $s =<:, 3, :>$ as a slice key is not possible. Felxible slicing mush alwasy be maintained to enable resolving triple patterns (Sec. ???). Tentris uses *Hypertrie*, a novel trie-based data structure, for holding the keys of an RDF tensor. Hypertrie generalize the normal trie tensor concept by enabling the efficient slicing by any combination of dimensions. Hypertrie is defined formally based on Tentris paper [7] in the following:

Definition 2.5. Let $H(d, A, E)$ with $d \geq 0$ be the set of all hypertries with depth d, alphabet A and values E. If A and E are clear from the context, we use $H(d)$. We set $H(0) = E$ per definition. A hypertrie $h \in H(1)$ has an associated partial function $c_1^{(h)} : A \nrightarrow E$ that specifies outgoing edges by mapping edge labels to children. For $h^{'} \in H(n), n > 1$, partial functions $c_i^{(h')} : A \nrightarrow H(d-1), i \in \mathbb{N}_n$ are defined. Function $c_i^{(h')}$ specifies the edges for resolving the part equivalent to depth i in a trie by mapping edge labels to children. For a hypertrie h, $z(h)$ is the size of the set or mapping it encodes.

Continue like this Informally, a .....

Figure 2.5 shows a hypertrie storing the RDF triples shown in table 2.1. Examples for slicing hypertrie with different slice keys corresponding to various combinations of slicing positions is presented in figure 2.6.

add definition here as needed
Add the concept of node $processed\_nodes : \mathbb{B}^3 \rightarrow node$

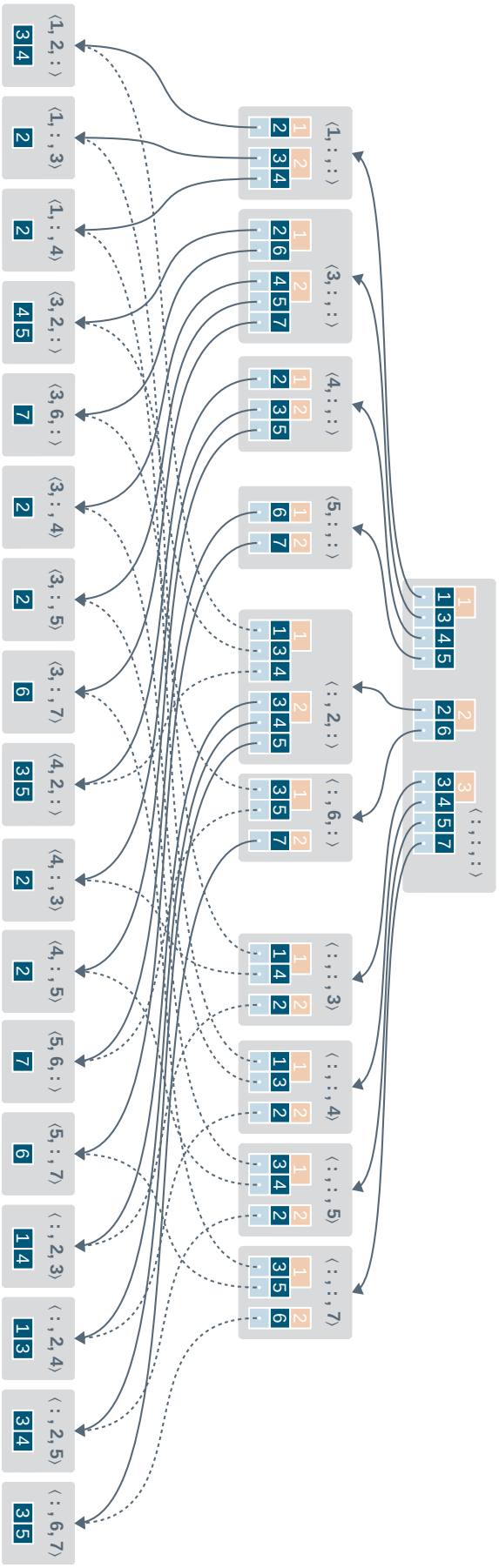


Figure 2.5: Trie representation of the tensor T_g that depicts the RDF graph g in Table 2.3. A slice $T_g[3,:,:]$ by the first dimension with 3 is shown in the red (inner) box.

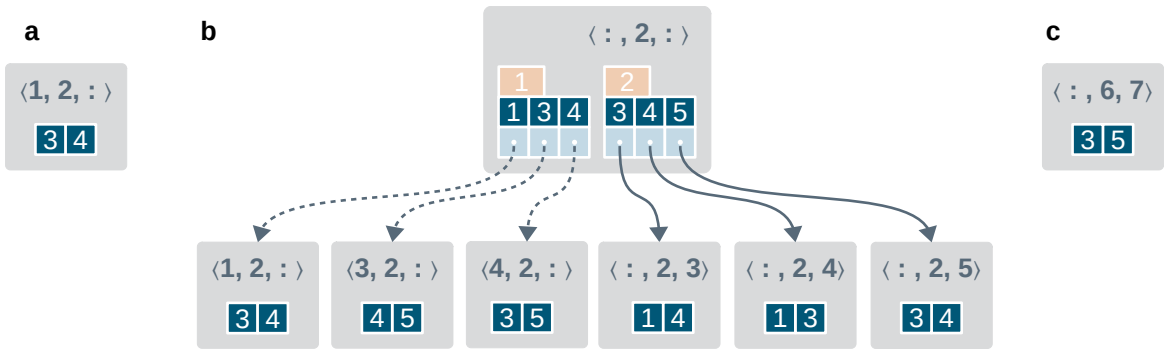


Figure 2.6: Slice.

3

Related Work

The efficiency of in-memory data structures used for storing and retrieving strings (or other sequence types) is crucially essential for applications such as in-memory data store implementation. Trie-based data structures offer rapid access to strings while maintaining good worst-case performance. However, they are space-intensive. Consequently, measures need always to be taken to reduce their space consumption if they are to remain feasible for maintaining large sets of sequences.

Burst Trie has been the most successful procedure for reducing the space of a trie structure [14]. It selectively collapses chains of trie nodes into small containers of strings that share a common prefix. By that, burst trie reduces the number of trie nodes up to 80%. When a container has reached its capacity, it is burst into smaller containers parented by a new trie node. Burst trie uses a linked list to chain its nodes. Its concept is based on the burst-sort [22].

HAT-trie [3] builds on top of burst trie and establishes the idea of chaining tire nodes using hash tables instead of a linked list in an attempt to deliver a more cache-conscious in-memory string data structure.

Adaptive Radix Tree (ART) [17] presents a novel, efficient and space-friendly trie-based data structure. In their work [17], ART introduces the concept of adaptive nodes. Depending on the size of data held by the node, local representation is assigned for each node from a pre-defined set of inner node data structures.

PATRICIA [18] is an approach to minimize the amount of memory space consumed by tries holding long sequences of strings.

4

Space Reduction Approach

In this chapter, I discuss the adopted approach to substantially mitigates the space inefficiency 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's memory footprint. In contrast, it could be considered an added feature that further contributes to the space reduction of the overall Hypertrie data structure.

This part of the work delivers motivation to the approach. Afterward, an illustration of the new Hypertrie internal nodes' design needed to realize the solution is delivered. The chapter ends with a presentation of the cor algorithms that define the behaviors of the newly designed Hypertrie.

4.1 Hypertrie Memory Utilization

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 [8].

The current design and implementation of Hypertrie [7], however, mitigates the space inefficiency characteristic in two ways. First, each partial function $c_i^{(h)}$ (Def. 2.5) in each hypertrie node h is realized using a custom sparse hash table HT instead of arrays or linked lists to store the key parts. By using hash table, 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 (Sec. 2.4.3).

The adoption of hash tables in Hypertrie also delivers extra performance as looking up keys is nearly constant compared to linked list search where it has a linear computational complexity $O(n)$. The other attempt to reduce the overall space requirement is to store equal nodes (Sub-hypertrie) only once by making use of the associativity characteristic of the slicing operations in tensors (Sec. 2.4.1). That is, hypertrie takes advantage of that fact that the slicing order relative to the root node has no influence when slices are chained. Example: $T[k_0, k_1, ...] = (T[k_0, :, ...])[k_1, ...] = (T[:, k_1, ...])[k_0, ...]$.

By exploiting the equivalent slicing concept and store equal sub-hypertrie nodes only once, it is proved that the upper storage bound of hypertrie $h \in H(d)$ is $\vartheta(2^{d-1}.d.z(h))$, where $z(h)$ is the number of non-zero entries [7]. Since, hypertrie realizes an RDF tensor, then $d = 3$. Thus, the upper storage bound is $\vartheta(12.z(h))$.

4.2 Space Reduction Solution

Despite the previously mentioned attempts to minimize the size of Hypertrie (Sec. 4.1), the excessive memory utilization is still a bottleneck. The case can be witnessed in practice when the set of RDF triples needed to be indexed by hypertrie increases in size with less overlapping between its triples. As a result, a set of nodes, including the root node, will store edges to children with single entry for a particular dimension. Nevertheless, we still need to utilize a space to host a pointer to that child node. This results in a space redundancy issue [18].

Example 4.1. Back to the example depicted in figure 2.5:

1. The inner node labeled with `<4, :, :>` contains an edge = 3 at position 1 that maps to leaf node with a single entry.

2. Another example in the same structure is the root node r along with its child $h = c_0^{(r)}(5)$. It is noticeable that h also map its edges to single-element lead nodes.

Inspired by PATRICIA [18] and ART [17], the redundancy problem mentioned above can be eliminated by recursively storing a single-element children into their parent nodes resulting in a *path compression*. In other words, we compress the hypertrie branches by storing *key suffixes* into their parent node at the key part from which the suffix was branched. Back to the example 4.1, the leaf node labeled with `<4, :, 3>` can be compacted along with its parent edge in its parent node `<4, :, :>` at the edge 3. Similarly, the leaf nodes `<5, 6, :>` and `<5, :, 7>` can be stored into their parent node `<5, :, :>` at the respective edges. However, the parent node itself represents a single sub key, hence it can be compacted into the root node at edges = 5 in position 0. Worth to mention that applying the presented path compression approach doesn't change the space complexity class of hypertrie, however, it greatly minimize the memory utilization in practice.

Applying the path compression approach this way will introduce a modified version of hypertrie, a *compressed hypertrie*. Edges that map to key suffixes stored in parent nodes are termed as *compressed edges*. Figure 4.1 depicts a compressed hypertrie variant that stores the RDF graph g in table 2.1. when the path compression technique is applied. It is noticeable 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 allocating extra space to store pointers to children.

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. As the size of key suffixes varies depending on the depth of the parent node, each group of Hypertrie nodes at certain depth will have their own internal node representation. In details, each hypertrie node of depth

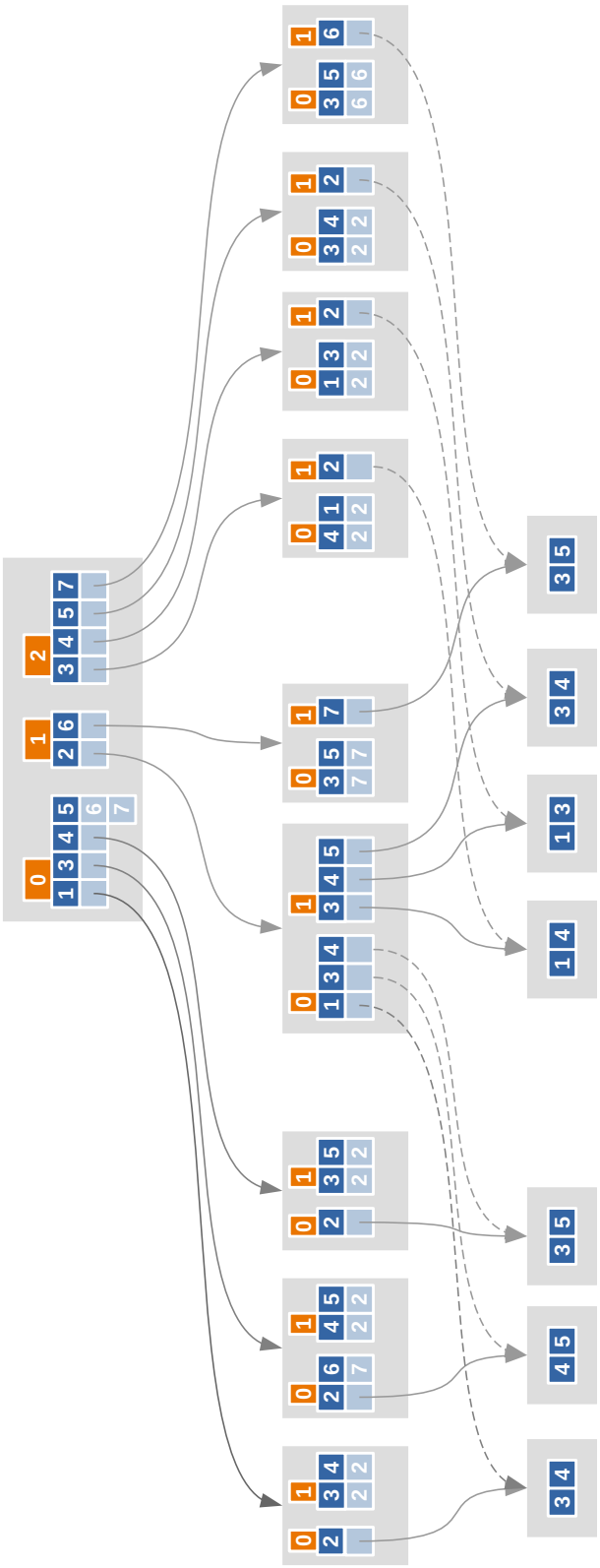


Figure 4.1: Storing RDF in space-efficient Hypertrie (Compressed Hypertrie).

d should realize a *container* concept of size $d-1$ for every compressed path stored in it. On the other hand, compressed hypertrie nodes should be *adaptive* to change. As the hypertrie state changes, for example by adding a new key, holding a compressed key path at certain edge may no longer be valid. The node at that particular edge mush be branched down to child nodes.

From programming point of view, the redesign of Hypertrie nodes' structures is low level. Thanks to C++17 template meta-programming features, 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. In this section, I delve into the internal design of each type of compressed hypertrie nodes and discuss the concept of virtual node that enable slicing in the compressed mode.

Remark 4.1. The following sections make use of all the terminologies, definitions, and conventions defined in section 2.4 that refer to various aspects of tensors and trie (including hypertrie).

Remark 4.2. The term $H(d)$ in the following sections will refer to the set of all nodes of depth d in the compressed version of hypertrie, not the original hypertrie.

4.3.1 Internal Node Representations

This section summarizes the design decision of compressed hypertrie nodes. A major step to implement the path compression approach is to the container concept for each node type[1] (Sec. 4.3). 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 stored in containers bounded to other edges (compressed edges).

The compressed key path container implementation varies depending on the node depth. 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_p for each position $p \in P = \{0, 1, 2\}$, the root node $r \in H(d = 3)$ additionally holds another array of hash tables $CommHT_p$ that maps key parts k_j to static arrays arr_j of the size $d-1 = 2$. 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. In this context, the key part k_j stored in $CommHT_p$ represent a compressed edge. 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. 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.

Depth 2 Node

Internal nodes $h_2 \in H(2)$ realize the static container concept associated with compressed edges differently than for the root node. Considering the number of internal nodes, it becomes unfeasible to assign extra set of hash tables for each node that serve as containers for key part chains.

To implement the container concept, the design of depth 2 nodes exploits the fact that key suffixes for edges 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 (Sec. 2.2).

Consequently, pointers to children that corresponds to edges k_j in h_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.

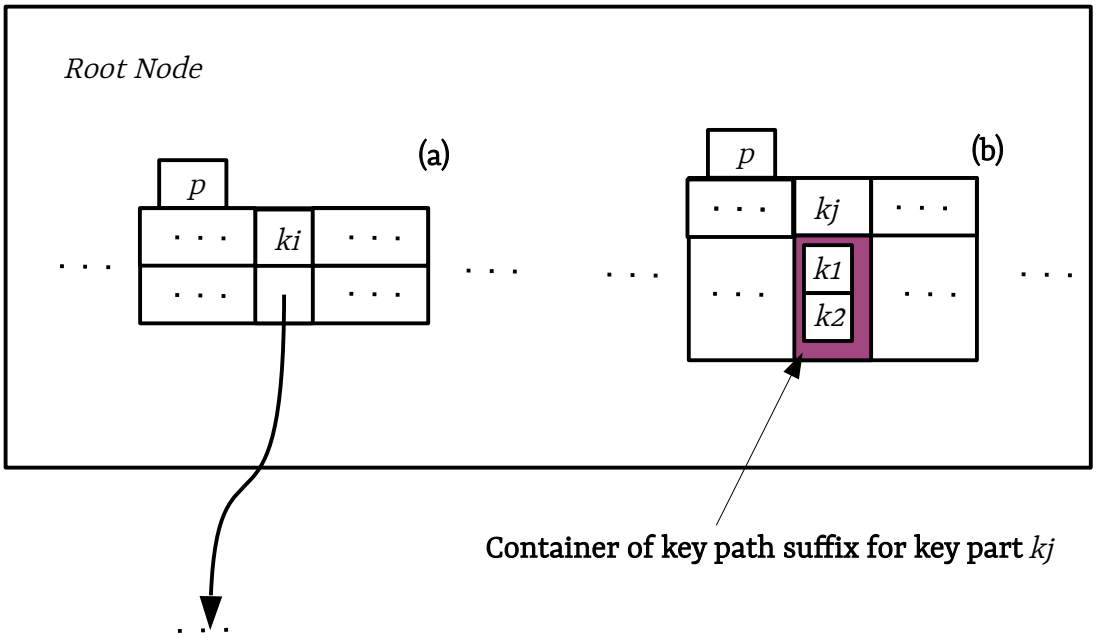


Figure 4.2: Depth 3 Node

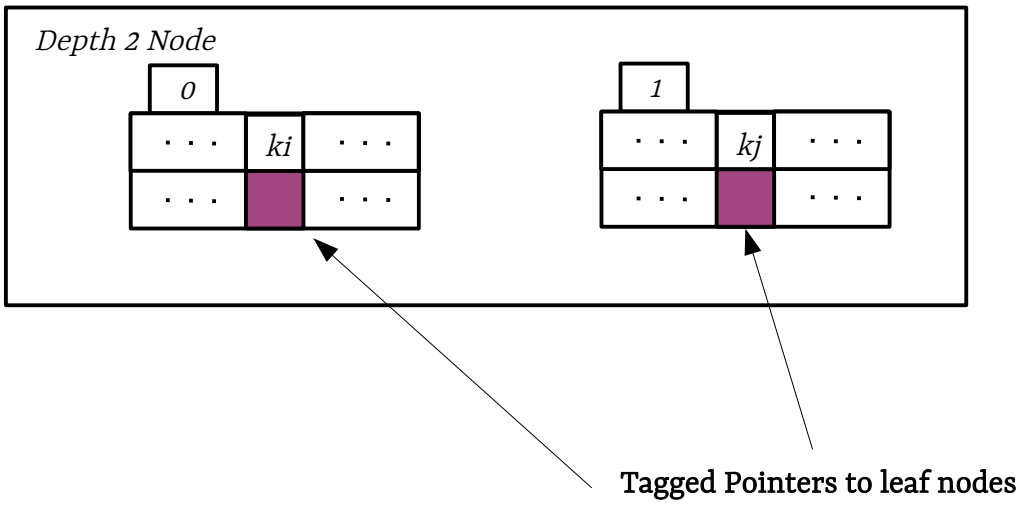


Figure 4.3: Depth 2 Node

4.3.2 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 4.3. 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 4.1: 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 4.2 shows the structure of the tagged pointer I developed

to reference different node types in our space-friendly Hypertrie.

Listing 4.2: 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 4.3: 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].

4.4.1 Key Retrieval

A basic operation in Hypertrie is to check if a given key $k \in N^d$ represents a path from a hypertrie node $h \in H(d)$ to a leaf node. 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 type.

depth 3 Node

depth 2 Node

depth 1 Node

[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

```
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

Input: *node*1: a depth 1 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  node1_key_retrieval(node1, depth, key)
2  begin
3  |  return set of edges in node1 holds the key part key[0]
4  end
```

Algorithm 3: Key Retrieval in depth 1 Nodes

4.4.2 Key Insertion

Similar to other tree-based data structures, the key insertion operation is recursive. Due to the fact that nodes at certain depth enjoys a unique internal structure, the algorithm for adding keys will change depending on the target node in the recursion sequence. To achieve key insertion, two auxiliary constructs must be implemented. First, the algorithm should utilize a mechanism that keeps track of the processed key parts so far for each key path in hypertrie. A way to maintain such state information is to use a tuple $processed \in \mathbb{B}^3$, such that:

$$\forall i \in \mathbb{N}_3, processed[i] = \begin{cases} 1, & \text{key part at position } i \text{ is processed} \\ 0, & \text{otherwise} \end{cases}$$

Utilizing the state tuple *processed* while processing a hypertrie node, we know the *sub key*, the remaining key parts to be added. Hence, the state tuple is updated as we proceed in the construction of the key path during insertion; i.e. processing more key parts.

For that, a couple of utility functions are used to map the main key to sub keys. That is, they map key part indices of the inserted key to dimensions represented by the succeeding child node and vise versa. By that, we know which key part should be inserted at which position. *pos*2*ids* maps node positions to key part indices, and *id*2*pos* maps key part indices to node positions. Both functions are defined in the following:

$$pos2id(pos, processed) = pos + |\{e | e \in \text{dom}(processed), e \leq pos, processed[e] = 1\}|$$

$$id2pos(id, processed) = id - |\{e | e \in \text{dom}(processed), e \leq id, processed[e] = 1\}|$$

The second requirement to achieve efficient insertion similar to the original hypertrie insertion method is to preserve the concept of pointing to equal child nodes only once. This is done by setting up a dictionary $processed\_nodes : \mathbb{B}^3 \rightarrow node$ for each key insertion call. *processed_nodes* maps instances of the state tuple *processed* to inform which hypertrie nodes were already created for which combination of dimensions.

Input: *node*3: a depth 3 node, *depth*: current node depth, *key*: array of key parts

```
1 insert(node3, depth, key)
2 begin
3 |   processed ←< 0, 0, 0 >
4 |   processed_nodes ← {}
5 |   node3_insert(node3, depth, key, processed, processed_nodes)
6 end
```

Algorithm 4: Key insertion main method

Input: *node*3: a depth 3 node, *depth*: current node depth, *key*: array of key parts, *processed*: state of processed key parts, *processed_nodes*: dictionary of processed nodes

```
1  node3_insert(node3, depth, key, processed, processed_nodes)
2  begin
3      foreach key_pos in {e|e ∈ dom(processed), processed[e] = 0} do
4          key_part ← key[key_pos]
5          pos ← id2pos(key_pos, processed)
6          next_processed = processed
7          next_processed[key_pos] ← 1
8          child_ptr ← get(node3, pos, key_part)
9          (child, tag) ← child_ptr
10         if child is empty then
11             arr ← [0, 0]
12             for i in [0, 1] do
13                 arr[i] = key[pos2id(i, next_processed)]
14             end
15             HTComm_pos[key_part] = arr
16         else if tag == NON_COMPRESSED then
17             node2_insert(child, 2, key, next_processed, processed_nodes)
18         else if tag == COMPRESSED then
19             new_node ← create a new concrete hypertrie child node of depth 2
20             HT_pos[key_part] = ptr(new_node)
21             node2_insert(new_node, 2, key, next_processed, processed_nodes)
22             reconstruced_key ←< 0, 0, 0 >
23             arr ← HTComm_pos[key_part]
24             for i in [0, 1, 2] do
25                 if i == key_pos then
26                     reconstruced_key[i] = key_part
27                 else
28                     reconstruced_key[i] = arr[id2pos(i, next_processed)]
29                 end
30             end
31             erase HTComm_pos[key_part]
32             node2_insert(new_node, 2, reconstruced_key, next_processed, processed_nodes)
33         end
34     end
35 end
```

Algorithm 5: Key insertion for depth 3 node

Input: *node*2: a depth 2 node, *depth*: current node depth, *key*: array of key parts, *processed*: state of processed key parts, *processed_nodes*: dictionary of processed nodes

```
1  node2_insert(node2, depth, key, processed, processed_nodes)
2  begin
3      processed_nodes[processed] ← node2
4      foreach key_pos in {e|e ∈ dom(processed), processed[e] = 0} do
5          key_part ← key[key_pos]
6          pos ← id2pos(key_pos, processed)
7          next_processed ← cloning processed
8          next_processed[key_pos] ← 1
9          child_ptr ← HTpos[key_part]
10         (value, tag) ← child_ptr
11         if child_ptr is empty then
12             next_key_part = key[pos2id(0, next_processed)]
13             HT_pos[key_part] ← tagged pointer holding next_key_part integer
14         else
15             if tag = IS_POINTER then
16                 child_node ← node to which address in value points
17                 node1_insert(child_node, 1, key, next_processed, processed_nodes)
18             else if tag = IS_INT then
19                 created_node ← processed_nodes[next_processed]
20                 if created_node exists then
21                     value ← hold a pointer to created_node
22                     tag ← IS_POINITER
23                 else
24                     reconstruced_key ←< 0, 0, 0 >
25                     for i in [0, 1, 2] do
26                         if processed[i] == 1 then reconstruced_key[i] = key[i]
27                         else if i == key_pos then reconstruced_key[i] = key_part
28                         else reconstruced_key[i] = value
29                     end
30                 end
31                 new_node ← create a new concrete hypertrie child node of depth 1
32                 HT_pos[key_part] ← tagged pointer holding pointer to new_child
33                 node1_insert(new_node, 1, key, reconstruced_key, next_processed, processed_nodes)
34             end
35         end
36     end
37 end
```

Algorithm 6: Key insertion for depth 2 node

Input: *node*1: a depth 1 node, *depth*: current node depth, *key*: array of key parts, *processed*: state of processed key parts, *processed_nodes*: dictionary of processed nodes

```
1 node1_insert(node1, depth, key, processed, processed_nodes)
2 begin
3 |  created_node ← processed_nodes[processed]
4 |  if created_node does not exist then
5 |  |  processed_nodes[processed] = node1
6 |  end
7 |  key_part ← key[pos2id(0, processed)]
8 |  add key_part to the set of key parts (edges) held by node1
9 end
```

Algorithm 7: Key insertion for depth 1 node

Input: *node*1: a depth 1 node, *depth*: current node depth, *key*, *reconstruced_key*: array of key parts,
processed: state of processed key parts, *processed_nodes*: dictionary of processed nodes

```
1 node1_insert(node1, depth, key, processed, reconstruced_key, processed_nodes)
2 begin
3 |  created_node ← processed_nodes[processed]
4 |  if created_node does not exist then  processed_nodes[processed] = node1
5 |
6 |  key_part_1 ← key[pos2id(0, processed)]
7 |  key_part_2 ← reconstruced_key[pos2id(0, processed)]
8 |  add key_part and key_part_2 to the set of key parts (edges) held by node1
9 end
```

Algorithm 8: Key insertion for depth 1 node including the insertion of the reconstructed key

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 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 4 WatDiv synthetic data corresponding to scale factors (10, 100, 1000, 10000) [2]. 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 adopted to calculate the load speed of RDF graphs 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 time 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 [9]. 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

Data set	# tripls	size (kB) (Compressed)	size (kB) (non-Compressed)	CR
WatDiv (scale=10)	1097093	107148	422744	3.945
WatDiv (scale=100)	10985185	1032960	4191216	4.057
WatDiv (scale=1000)	109911677	10289024	41760552	4.059
WatDiv (scale=10000)	1098338594	104048444	413277780	3.972
DBpedia	1087348475	97680540	361838396	3.704
SWDF	372397	71956	205092	2.85

Table 5.2: Results of size calculation.

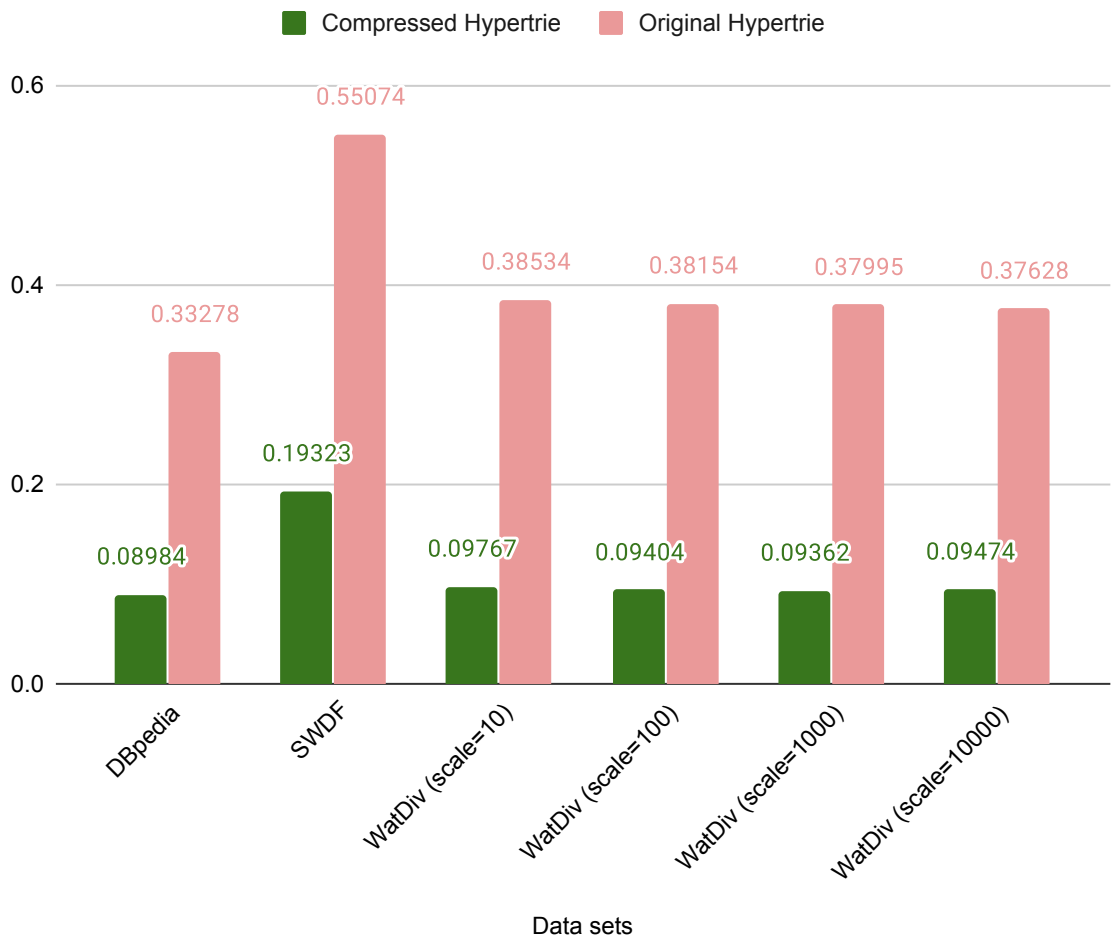


Figure 5.1: The bar chart shows the average size (in kB) acquired by a single triple for each data set when it is stored in either variants of Hypertrie.

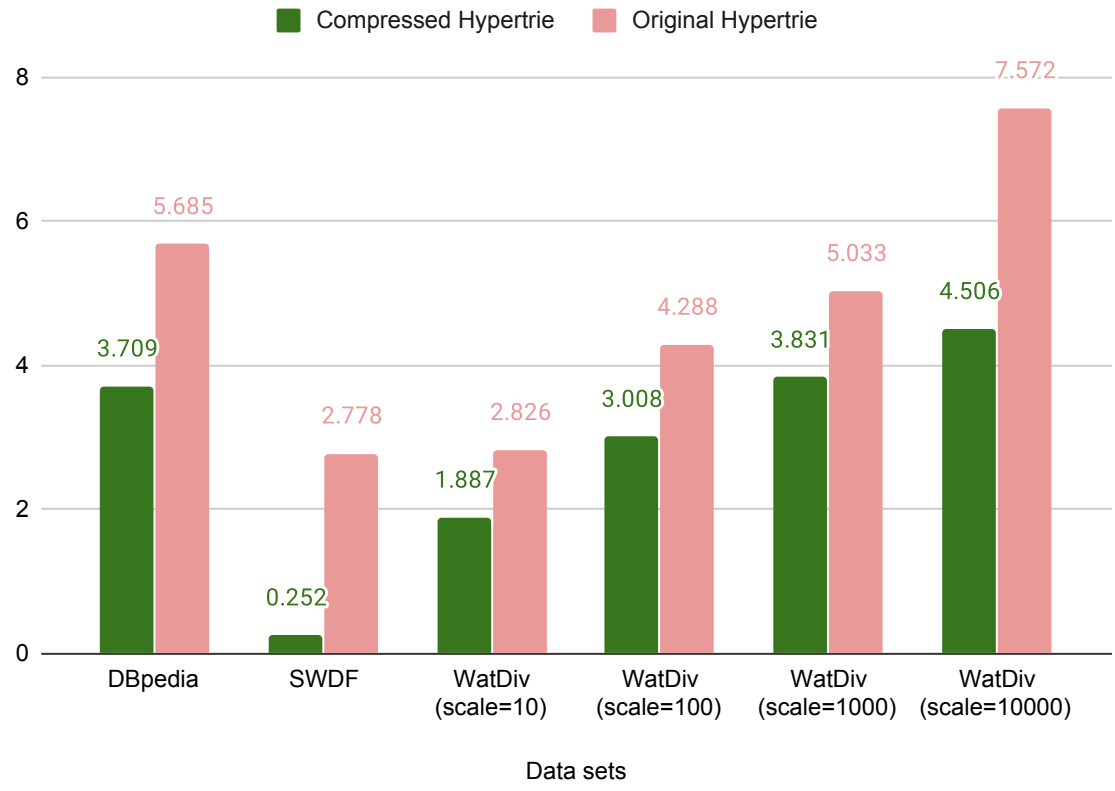
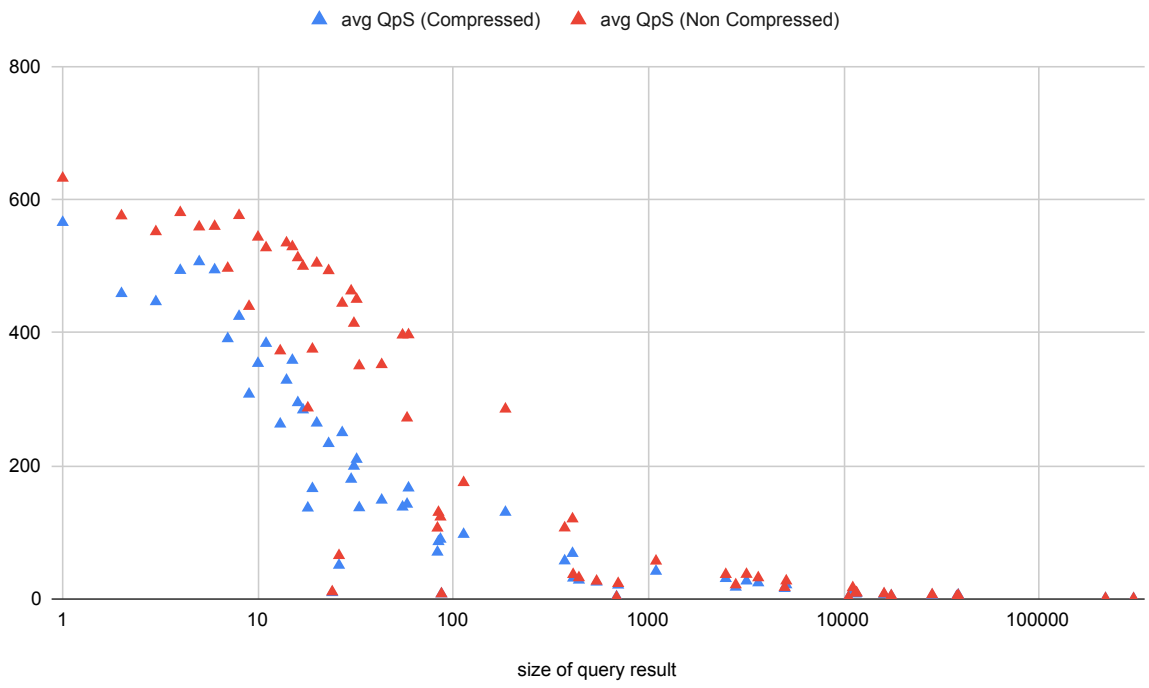


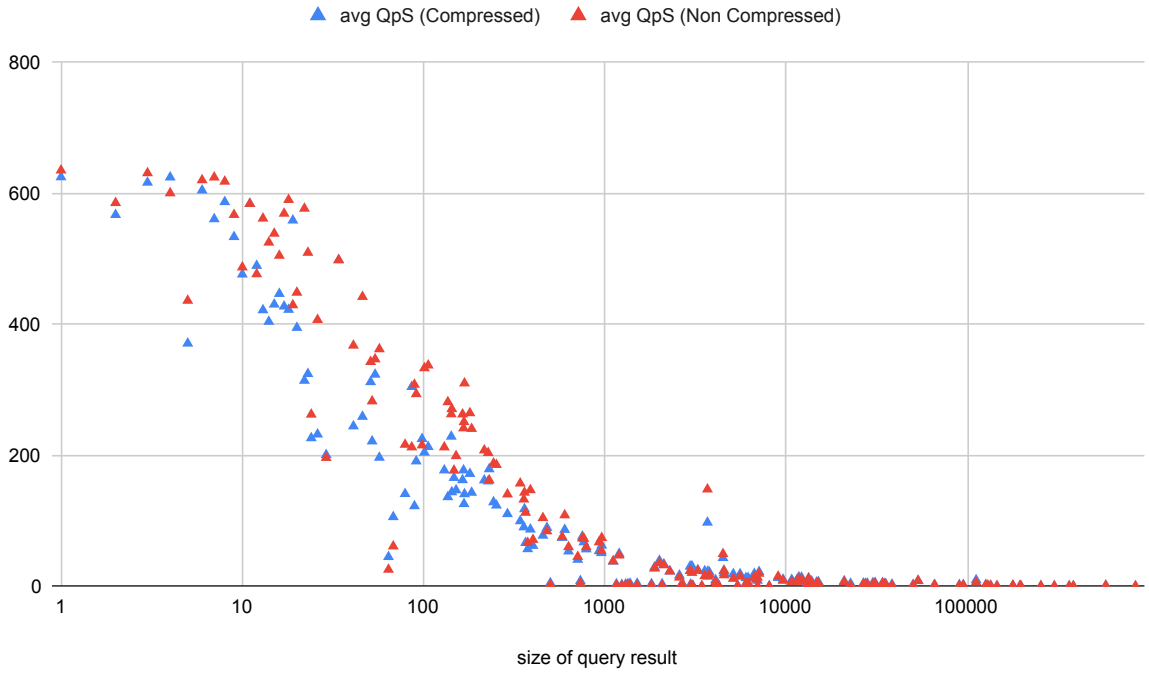
Figure 5.2: The average load time (in μs) of an RDF triple calculated during the bulk loading experiment.

Data set	# tripls	load time (sec) (Compressed)	load time (sec) (non-Compressed)
WatDiv (scale=10)	1097093	2.0702	3.0998
WatDiv (scale=100)	10985185	33.0395	47.0945
WatDiv (scale=1000)	109911677	421.0285	553.088
WatDiv (scale=10000)	1098338594	4948.0489	8316.0049
DBpedia	1087348475	4032,0386	6181,0737
SWDF	372397	0.0935	1.0345

Table 5.3: Results of load speed calculation.



(a) SWDF



(b) DBpedia

Figure 5.3: 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 results of benchmarking against SWDF dataset and associated queries, while figure 5.1b shows the results of benchmarking against DBpedia dataset.

6

Discussion and Future Work

Parallelizeation Different types of containers Different types of compression methods

Bibliography

[1] World Wide Web Consortium. RDF Current Status - W3C. `https://www.w3.org/TR/?tag=data/`. (visited: 07-Sep-2020).

[2] Aluç, G., Hartig, O., Özsu, M. T., and Daudjee, K. Diversified stress testing of rdf data management systems. In *Proceedings of the 13th International Semantic Web Conference - Part I* (Berlin, Heidelberg, 2014), ISWC '14, Springer-Verlag, p. 197–212.

[3] Askitis, N., and Sinha, R. Engineering scalable, cache and space efficient tries for strings. *The VLDB Journal 19*, 5 (Oct. 2010), 633–660.

[4] Atre, M., Chaoji, V., Zaki, M., and Hendler, J. Matrix "bit" loaded: a scalable lightweight join query processor for rdf data. pp. 41–50.

[5] Auer, S., Lehmann, J., Ngonga Ngomo, A.-C., and Zaveri, A. Introduction to linked data and its lifecycle on the web. In *Proceedings of the 9th International Conference on Reasoning Web: Semantic Technologies for Intelligent Data Access* (Berlin, Heidelberg, 2013), RW'13, Springer-Verlag, p. 1–90.

[6] Berners-Lee, T., Fischetti, M., and Dertouzos, M. L. *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web by Its Inventor*, 1st ed. Harper San Francisco, 1999.

[7] Bigerl, A., Conrads, F., Behning, C., Sherif, M., Saleem, M., and Ngonga Ngomo, A.-C. Tentris – A Tensor-Based Triple Store. In *International Semantic Web Conference* (2020).

[8] Brass, P. *Advanced Data Structures*, 1 ed. Cambridge University Press, New York, NY, USA, 2008.

[9] Conrads, F., Lehmann, J., Saleem, M., and Ngomo, A.-C. N. Benchmarking rdf storage solutions with iguana. In *Proceedings of 16th International Semantic Web Conference - Poster & Demos* (2017).

[10] De Virgilio, R. A linear algebra technique for (de)centralized processing of sparql queries. In *Conceptual Modeling* (Berlin, Heidelberg, 2012), P. Atzeni, D. Cheung, and S. Ram, Eds., Springer Berlin Heidelberg, pp. 463–476.

[11] Duerst, M., and Suignard., M. Internationalized Resource Identifiers (IRIs). `https://www.ietf.org/rfc/rfc3987.txt`. Internet Engineering Task Force, Jan. 2005. (visited: 10-Sep-2020).

[12] Einstein, A. Die grundlage der allgemeinen relativitätstheorie. *Annalen der Physik 354*, 7 (1916), 769–822.

[13] Erling, O. Virtuoso, a Hybrid RDBMS/Graph Column Store. `http://vos.openlinksw.com/owiki/wiki/VOS/VOSArticleVirtuosoAHybridRDBMSGraphColumnStore`. (visited: 30-Aug-2020).

[14] Heinz, S., Zobel, J., and Williams, H. Burst tries: A fast, efficient data structure for string keys. *ACM Trans. Inf. Syst. 20* (04 2002), 192–223.

[15] Hitzler, P., Krtzsch, M., and Rudolph, S. *Foundations of Semantic Web Technologies*, 1st ed. Chapman & Hall/CRC, 2009.

[16] Kroes, T., Koning, K., Kouwe, E., Bos, H., and Giuffrida, C. Delta pointers: buffer overflow checks without the checks. pp. 1–14.

[17] Leis, V., Kemper, A., and Neumann, T. The adaptive radix tree: Artful indexing for main-memory databases. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)* (USA, 2013), ICDE '13, IEEE Computer Society, p. 38–49.

[18] Morrison, D. R. Patricia—practical algorithm to retrieve information coded in alphanumeric. *J. ACM 15*, 4 (Oct. 1968), 514–534.

[19] Neumann, T., and Weikum, G. The rdf-3x engine for scalable management of rdf data. *The VLDB Journal 19*, 1 (Feb. 2010), 91–113.

[20] Renteln, P. *Manifolds, Tensors and Forms.* Cambridge University Press, 2014.

[21] Silberschatz, A., Galvin, P. B., and Gagne, G. *Operating System Concepts*, 9th ed. Wiley Publishing, 2012.

[22] Sinha, R., and Zobel, J. Efficient trie-based sorting of large sets of strings. In *ACSC* (2003).

[23] Stroustrup, B. *The C++ Programming Language*, 4th ed. Addison-Wesley Professional, 2013.

[24] SYSTAP, LLC. Bigdata Database Architecture - Blacegraph. `https://www.blazegraph.com/whitepapers/bigdata_architecture_whitepaper.pdf`. (visited: 30-Aug-2020).

[25] The Apache Software Foundation. Apache Jena TDB. `http://jena.apache.org/documentation/tdb/`. (visited: 07-Aug-2020).

[26] Wood, D., Lanthaler, M., and Cyganiak, R. RDF 1.1 Concepts and Abstract Syntax, 2014. http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/.