Hashing the Hypertrie: Space- and Time-Efficient Indexing for SPARQL in Tensors

Alexander Bigerl[1(✉)], Lixi Conrads[1], Charlotte Behning[2], Muhammad Saleem[3], and Axel-Cyrille Ngonga Ngomo[1]

[1] DICE Group, Department of Computer Science, Paderborn University, Paderborn, Germany
{alexander.bigerl,axel.ngonga}@upb.de
[2] Department of Medical Biometry, Informatics and Epidemiology, University Hospital Bonn, Bonn, Germany
behning@imbie.uni-bonn.de
[3] CS Department, Leipzig University, Leipzig, Germany
saleem@informatik.uni-leipzig.de
https://dice-research.org/, https://www.imbie.uni-bonn.de/, https://www.mathcs.uni-leipzig.de/ifi

Abstract. Time-efficient solutions for querying RDF knowledge graphs depend on indexing structures with low response times to answer SPARQL queries rapidly. Hypertries—an indexing structure we recently developed for tensor-based triple stores—have achieved significant runtime improvements over several mainstream storage solutions for RDF knowledge graphs. However, the space footprint of this novel data structure is still often larger than that of many mainstream solutions. In this work, we detail means to reduce the memory footprint of hypertries and thereby further speed up query processing in hypertrie-based RDF storage solutions. Our approach relies on three strategies: (1) the elimination of duplicate nodes via hashing, (2) the compression of non-branching paths, and (3) the storage of single-entry leaf nodes in their parent nodes. We evaluate these strategies by comparing them with baseline hypertries as well as popular triple stores such as Virtuoso, Fuseki, GraphDB, Blazegraph and gStore. We rely on four datasets/benchmark generators in our evaluation: SWDF, DBpedia, WatDiv, and WikiData. Our results suggest that our modifications significantly reduce the memory footprint of hypertries by up to 70% while leading to a relative improvement of up to 39% with respect to average Queries per Second and up to 740% with respect to Query Mixes per Hour.

1 Introduction

The hypertrie [6], a monolithic indexing data structure based on tries, is designed to support the efficient evaluation of basic graph patterns (BGPs) in SPARQL. While the access order for the positions of the tuples in tries is fixed, a hypertrie allows to iterate or resolve tuple positions in arbitrary order. In previous work [6] we showed that hypertries of depth 3 are both time- and memory-efficient when combined with a worst-case optimal join (WCOJ) based on the Einstein summation algorithm. With the benchmarking of our implementation, dubbed TENTRIS, we also showed that hypertries

U. Sattler et al. (Eds.): ISWC 2022, LNCS 13489, pp. 57–73, 2022.
https://doi.org/10.1007/978-3-031-19433-7_4

outperform mainstream triple stores significantly on both synthetic and real-world benchmarks when combined with WCOJs. We analyzed the space requirements of hypertries on four RDF datasets: Semantic Web Dog Food (SWDF), DBpedia 2015-10, WatDiv, and Wikidata (see Sect. 5 for details on the datasets) revealing the following limitations of the current implementation: (1) The hypertries contain a high proportion of duplicate nodes, i.e. between 72% (SWDF) and 84% (WatDiv) (see baseline vs. hash identifiers in Fig. 1)

Two main conclusions can be derived from this analysis. First, the duplicate nodes lead to an unnecessarily high memory footprint. The addition of deduplication to hypertries could hence yield an improved data structure with lower memory requirements. Second, the high number of single-entry nodes might lead to both unnecessary memory consumption and suboptimal query runtimes. A modification of the data structure to accommodate single-entry nodes effectively has the potential to improve both memory footprint and query runtimes.

1. **Hash-Based identifiers (h)**: We modify the hypertrie to use hashes of nodes as primary keys. Hence, we store nodes with the same entries exactly once, thus eliminating duplicates.
2. **Single-Entry node (s)**: Single-entry nodes store the sub-hypertries of which they are the root node directly, thus saving space and eventually eliminating child nodes.
3. **In-Place storage (i)**: Boolean-valued single-entry nodes are eliminated completely.

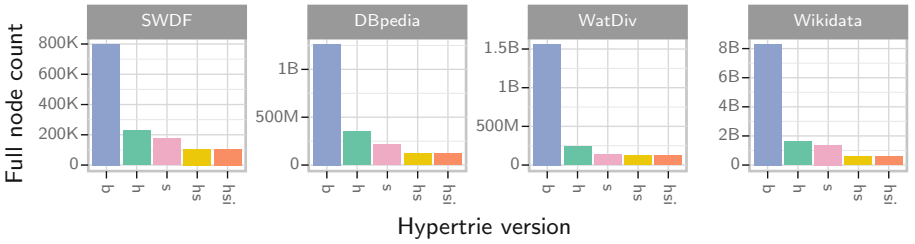


Fig. 1. Full node counts of different hypertrie versions on four datasets. The hypertrie versions are identified by their features: baseline (b), single-entry node (s), hash identifiers (h), and in-place storage of height-1 single-entry nodes (i).

The number of full nodes required by our optimizations is shown in Fig. 1. By applying all three techniques, the number of stored nodes is reduced by 82–90% (SWDF, WatDiv), and the memory consumption is reduced by 58–70% (SWDF, WatDiv), while the number of queries answered per second increases by up to four orders of magnitude on single queries.

The rest of this paper is structured as follows. First, we discuss related work in Sect. 2. In Sect. 3, we specify notations and conventions, introduce relevant concepts and describe the baseline hypertrie. We present our optimizations of the hypertrie in Sect. 4 and evaluate our optimized hypertries in Sect. 5. Finally, we conclude in Sect. 6.

2 Related Work

Many query engines for RDF graphs have been proposed in recent years [1,3,6,9–11,16,17,20,22]. Different engines deploy different mixes of indices and have different query execution approaches partly dependent on their indices. A common approach among SPARQL engines is to build multiple full indices in different collation orders such as Fuseki [10], Virtuoso [9], Blazegraph [20], and GraphDB [17]. Some systems build additional partial indices on aggregates such as RDF-3X [16], or cache data for frequent joins such as gStore [22] for star joins. Building more indexes provides more flexibility in reordering joins to support faster query execution, while fewer indexes accelerate updates and require less memory.

When it comes to worst-case optimal joins (WCOJs) [5], classical indexing reaches its limits as indices for all collation orders are required. A system that takes this approach is Fuseki-LTJ [11], which implements the WCOJ algorithm Leapfrog TrieJoin (LTJ) [21] within a Fuseki triple store with indices in all collation orders. Recent works also propose optimized data structures that provide more concise indices with support for WCOJs. Qdags [15] provide support for WCOJs based on an extension of quad trees. Redundancy in the quad tree is reduced by implementing it as a directed acyclic graph (DAG) and reusing equivalent subtrees. A Circle [3] stores Burrows-Wheeler-transformed ID triples in bent wavelet trees along with an additional index to encode the triples of an RDF graph. Both Qdag and Circle are succinct data structures that must be built at once and do not support updates. In their evaluation of Circle, Arroyuelo et al. showed that Qdag and Circle are very space efficient, and that Fuseki-LTJ and Circle answer queries faster than state-of-the-art triple stores such as Virtuoso and Blazegraph with respect to average and median response times. The Qdag performed considerably worse in the query benchmarks than all other systems tested.

The idea for single-entry node and in-place storage is based on path compression, a common technique to reduce the number of nodes required to encode a tree by storing non-branching paths in a single node. It was first introduced by Morrison in PATRICIA trees [14]. Using hashing for deduplication, like in the proposed hypertrie context for hypertrie nodes (see Sect. 4.1 for details) is inspired by previous works on pervasive computing [13]. The hypertrie that we strive to optimize in this paper is, like the Qdag, internally represented as a DAG. As with the Qdag, the DAG nature of the hypertrie reduces the space requirement from factorial to exponential by the tuple length. The reduction is accomplished by eliminating duplicates among equal subtrees.

3 Background

In this section, we briefly introduce the notation and conventions used in the rest of this paper. In particular, we give a brief overview of relevant aspects of RDF, SPARQL, and tensors. We also provide an overview of the formal specification of hypertries. More details can be found in [6].

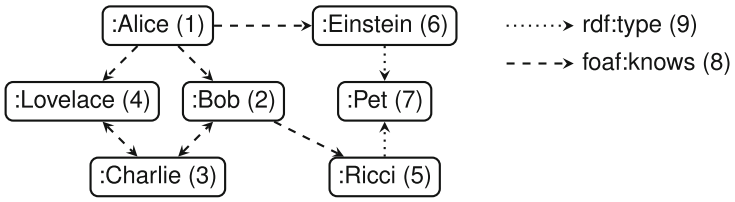


Fig. 2. Example RDF graph. Integer indices for RDF resources are provided in parentheses behind the string identifier.

3.1 Notation and Conventions

The conventions in this paragraph stem from [6]. Let $\mathbb{N}$ be the set of the natural numbers including 0. We use $\mathbb{I}_n := \{i \in \mathbb{N} \mid 1 \leq i \leq n\}$ as a shorthand for the set of natural numbers from 1 to n. The domain of a function f is denoted $\mathrm{dom}\,(f)$ while $\mathrm{cod}\,(f)$ stands for the target (also called codomain) of f. A function which maps x_1 to y_1 and x_2 to y_2 is denoted by $[x_1 \to y_1, x_2 \to y_2]$. Sequences with a fixed order are delimited by angle brackets, e.g., $l = \langle a, b, c \rangle$. Their elements are accessible via subscript, e.g., $l_1 = a$. The number of times an element e is contained in any bag or sequence C is denoted by $\mathit{count}\,(e, C)$; for example, $\mathit{count}\,(a, \langle a, a, b, c \rangle) = 2$. We denote the Cartesian product of S with itself i times with $S^i = \underbrace{S \times S \times \ldots S}_{i}$. We use the term *word* to describe a processor word, e.g. a 64-bit data chunk when using the x86-64 instruction set.

3.2 RDF and SPARQL

An RDF statement is a triple $\langle s, p, o \rangle$ and represents an edge $s \xrightarrow{p} o$ in an RDF graph g. s, p and o are called RDF resources. An RDF graph can be regarded as a set of RDF statements. The set of all resources of a graph g is given by $r(g)$. An example of an RDF graph is given in Fig. 2. The graph contains, among others, the RDF statement $\langle$:Alice, foaf:knows, :Bob$\rangle$.

A triple pattern (TP) Q is a triple that has variables or RDF resources as entries, e.g., $\langle$?x, foaf:knows, ?y$\rangle$. Matching a triple pattern Q with a statement t results in a set of zero or one solution mappings. If Q and t have exactly the same resources in the same positions, then matching Q to t results in a solution mapping which maps the variables of Q to the terms of t in the same positions. For example, imagine $Q = \langle$?x, foaf:knows, ?y$\rangle$ and $t = \langle$:Alice, foaf:knows, :Bob$\rangle$. Then $Q(t) = \{[$?x $\to$:Alice, ?x $\to$:Bob$]\}$. Otherwise, the set of solutions is empty, i.e., $Q(t) = \emptyset$. The result of matching a triple pattern Q against an RDF graph g is

$$Q(g) = \bigcup_{t \in g} Q(t), \tag{1}$$

i.e., the union of the matches of all triples t in g with Q. A list of triple patterns is called a basic graph pattern (BGP). The result of applying a BGP to an RDF graph g is the natural join of the solutions of its triple patterns.

Similar to previous works [4,6,16], we only consider the subset of SPARQL where a query is considered to consist of a BGP, a projection and a modifier (i.e., `DISTINCT`) that specifies whether the evaluation of Q follows bag or set semantics.

3.3 Tensors and RDF

Similar to [6], we use tensors that can be represented as finite multi-dimensional arrays. We consider a tensor of rank-n as an n-dimensional array $\mathbf{K}_1 \times \cdots \times \mathbf{K}_n \to \mathbb{N}$ with $\mathbf{K}_1 = \cdots = \mathbf{K}_n \subset \mathbb{N}$. Tuples from the tensor's co-domain $\mathbf{k} \in \mathbf{K}$ are called keys. The entries $\mathbf{k}_1, \ldots, \mathbf{k}_n$ of a key $\mathbf{k}$ are dubbed key parts. The array notation $T[\mathbf{k}] = v$ is used to express that T stores for key $\mathbf{k}$ value v.

The representation of g as a tensor, dubbed RDF tensor or adjacency tensor T, is a rank-3 tensor over $\mathbb{N}$ which encodes g. Let $id : r(g) \to \mathbb{I}_{|r(g)|+1}$ be an index function. The function maps each term of g—of which there are $|r(g)|$—to a fixed value in $\mathbb{I}_{|r(g)|}$. All unbound variables in solution mappings are mapped to $|r(g)|+1$. For all statements $\langle s, p, o\rangle \in g$, the value of $T[id(s), id(p), id(o)]$ is 1. All other values of T are 0. For example, $T[5, 9, 7] = 1$ for the example graph shown in Fig. 2, while $T[5, 9, 6] = 0$.

Matching a triple pattern Q against a graph g is equivalent to slicing the tensor representation of g with a slice key $\mathbf{s}(Q)$ corresponding to Q. The length of $\mathbf{s}(Q)$ is equal to the order of the tensor to which it is applied. Said slice key has a key part or a place holder, denoted ":" (no quotes), in every position. Slicing g with $\mathbf{s}(Q)$ results in a lower-order tensor that retains only entries where the key parts of the slice key match with the key parts of the tensor entries. For example, the slice key for the TP $Q = \langle$?x, foaf:knows, ?y$\rangle$ executed against the example graph in Fig. 2 is $\langle :, 8, :\rangle$. Applying the TP Q to g is homomorphic to applying the slice key $\mathbf{s}(t)$ to the tensor representation of g.

To define a tensor representation for sets or bags of solutions, we first define an arbitrary but fixed ordering function *order* for variables (e.g., any alphanumeric ordering). A tensor representation T' of a set or bag of solutions is a tensor of rank equal to the number of projection variables in the query. The index for accessing entries of T' corresponds to *order*. For example, given the TP $Q = \langle$?x, foaf:knows, ?y$\rangle$ with the projection variables $?x$ and $?y$, T' would be a matrix with $?x$ as the first dimension and $?y$ as the second dimension. After applying Q to the graph in Fig. 2, we would get a tensor T' with $T'[2, 5] = 1$ and $T'[5, 2] = 0$.

The Einstein summation [8,18] is an operation with variable arity. With this operation, the natural joins between the TPs of a BGP and variable projection can be combined into a single expression that takes the tensor representations of the TPs as input.

The execution of a SPARQL query on an RDF graph g is mapped to operations on tensors as follows. For each triple pattern, the RDF tensor T is sliced with the corresponding slice key. The slices are used as operands to an Einstein summation. Each slice is subscripted with the variables of the corresponding triple pattern. The result is subscripted with the projected variables. A ring with *addition* and *multiplication* is used to evaluate the Einstein summations. For example, evaluating the query with the BGP $\langle\langle$?x, foaf:knows, ?y$\rangle$, $\langle$?y, rdf:type, :Pet$\rangle\rangle$ and a projection to $?x$ on the RDF graph g from Fig. 2 is equivalent to calculating $\sum_x T[:, 8, :]_{x,y} \cdot T[:, 9, 7]_y$, where T is the RDF graph of g.

3.4 Hypertrie

A hypertrie is a tensor data structure that maps strings of fixed length d over an alphabet A to some value space V [6]. It is implemented as a directed acyclic graph to store tensors sparsely by storing only non-zero entries. Formally, [6, p.62] defines a hypertrie as follows:

Definition 1 (Hypertrie). *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 $c_p^{(h')} : A \nrightarrow H(d-1), p \in \mathbb{I}_n$ are defined. Function $c_p^{(h')}$ specifies the edges for resolving the part equivalent to depth p 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.*

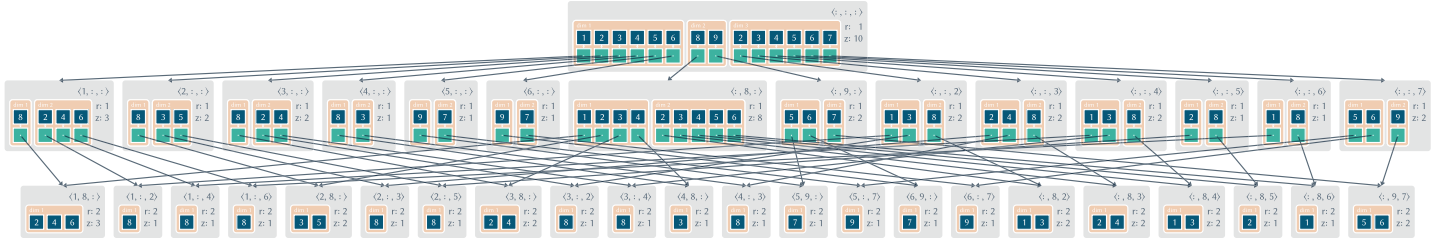
An example of a hypertrie encoding the RDF tensor of the graph in Fig. 2 is given in Fig. 3 with the baseline hypertrie.

To retrieve the value for a tensor key, we start at the root node. If the current node is from $H(0)$, it is the value and we are done. Otherwise, we select a key part from the key at an arbitrary position p. If c_p maps the selected key part, we descend to mapped sub-hypertrie, remove the selected key part from the key and repeat the retrieval recursively on the sub-hypertrie with the shortened key. Otherwise, the value is 0.

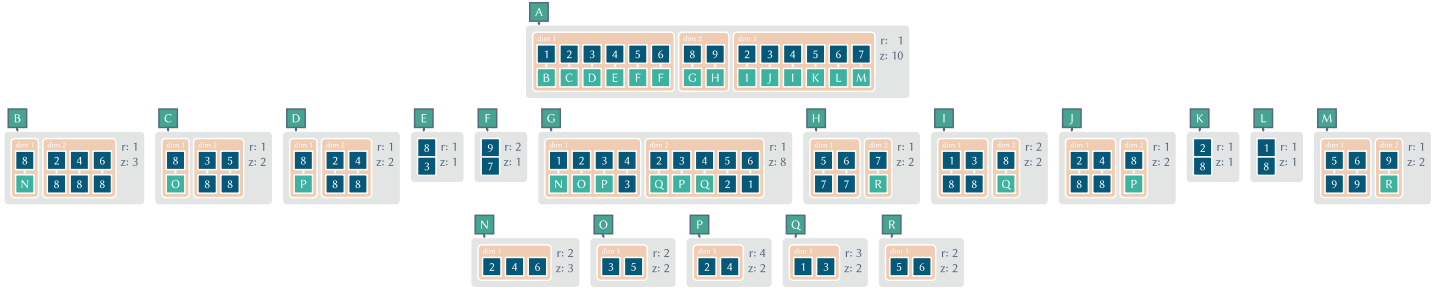
Hypertries are designed to satisfy four conditions: (R1) memory efficiency, (R2) efficient slicing, (R3) slicing in any order of dimensions, and (R4) efficient iteration through slices. [6] Furthermore, note that every hypertrie is uniquely identified by the set of tuples it encodes.

Implementation. We refer to the original implementation of the hypertrie [6] as baseline implementation. The baseline hypertrie is implemented in C++. The lifetime of hypertrie nodes is managed by reference-counting memory pointers which free the memory of a node when it is no longer referenced. For nodes with height $d > 1$, the edge mappings $c_{p \in \mathbb{I}_d}$ are stored in one hash table each. A node h' that is accessible from the root hypertrie h via different paths with equal slices is stored only once. Its parent nodes store a reference to the same physical instance of h'. For example, the slices $h[3, :, :][:, 4]$ and $h[:, :, 4][3, :]$ of a depth-3 hypertrie result in the same node. Nodes of depth $d = 1$ store the leaf edges in a hash set.

Hypertries were introduced as a tensor data structure for the tensor-based triple store TENTRIS. [6] In the following, we briefly describe the implementation of TENTRIS, which is later used to evaluate the improvements to the hypertrie presented in this paper. Consider an RDF graph g. A depth-3 Boolean-valued hypertrie is used by TENTRIS to store RDF triples encoded as integer triples. Therefore, the RDF resources $r(g)$ are stored as heap-allocated strings. The integer identifier of a resource is its memory address. We write $id(e)$ to denote the identifier of a resource e. id is implemented using a hash table while its inverse id^{-1} is applied by resolving the ID as memory address. Solutions of triple patterns are represented by pointers to sub-hypertrie nodes.



(a) Baseline hypertrie



(b) Hypertrie-hsi with all proposed optimizations

Fig. 3. Both hypertries encode the RDF graph from Fig. 2. RDF resources are encoded by their ID (see also Fig. 2).

Joins and projection are implemented with Einstein summation based on a worst-case optimal join algorithm.

4 Approach

In this section, we introduce three optimizations to the hypertrie. First, we eliminate duplicate nodes by identifying nodes with a hash. In a second step, we further reduce the memory footprint of hypertries by devising a more compact representation for nodes that encode only a single entry. Finally, we eliminate the separate storage of single-entry leaf nodes completely.

4.1 Hash-Based Identifiers

Our analysis of Fig. 1 suggests that equal sub-hypertries are often stored multiple times. To eliminate this redundancy, we first introduce a hashing scheme for hypertries that can be updated incrementally. Based thereupon, we introduce the *hypertrie context*, which keeps track of existing hypertrie nodes and implements a hash-based deduplication.

Hashing Hypertries. Let j be an order-dependent hashing scheme[1] for integer tuples. We define the hash i of a Boolean-valued hypertrie h as the result of applying j to the entries of h and aggregating them with XOR:

$$i(h) := \bigoplus_{\mathbf{k} \in \mathrm{dom}(h)} j(\mathbf{k}) \tag{2}$$

Since XOR is self-inverse, commutative, and associative, the hash can be incrementally updated by $i(h) \oplus j(\mathbf{k})$ when a key $\mathbf{k}$ is added or removed. Rather than rehashing and combining all entries again in $\mathcal{O}(z(h))$, the incremental update of the hash can be done in constant time. The hashing scheme can easily be extended to hypertries that store non-Boolean values by appending the value to the key before j is applied.

Hypertrie Context. The goal of a hypertrie context is to ensure that hypertrie nodes are stored only once, regardless of how often they are referenced. We now describe the design requirements for hypertrie contexts, provide a formal definition, and conclude with implementation considerations.

[1] In our implementation, we use the hash functions from https://github.com/martinus/robin-hood-hashing since preliminary experiments showed that they performed well and had no collisions on the datasets from Sect. 5.

In their baseline implementation, hypertrie nodes are retrievable by their path from the root of the hypertrie only. Information pertaining to the location of a node in memory is only available within its parent nodes. Consequently, only nodes with equivalent paths, i.e., with equal slice keys, are deduplicated in the baseline implementation. Equal hypertries with different slice keys are stored independently of each other. Hypertrie contexts eliminate these possible redundancies by storing hypertrie nodes by their hash and tracking how often nodes are referenced. The parent nodes are modified to reference their child nodes using hashes instead of memory pointers. Identifying hypertrie nodes by their hashes ensures that there are no duplicates.

A hypertrie can be *contained* or *primarily contained* in a hypertrie context hc. All nodes managed by a hypertrie context are *contained* therein. A hypertrie is said to be *primarily contained* in a hypertrie context hc iff it was stored explicitly in said context. For example, the root node of a hypertrie used for storing a given graph is commonly *primarily contained* in a hypertrie context. If a hypertrie h is *primarily contained* in a hypertrie context hc, then all sub-hypertries of h are *contained* in hc.

Adding a new primarily contained hypertrie or changing an existing hypertrie may alter the set of hypertries contained in a hypertrie context. To efficiently decide whether a node is still needed after a change, the hypertrie context tracks how often each node is referred to. Nodes that are no longer referenced after a change are removed. In *hypertrie contexts*, hypertries are considered to reference their sub-hypertries by hash.[2]

Formally, we define a *hypertrie context* as follows:

Definition 2 (Hypertrie Context). *Let A be an alphabet, E a set of values, and $d \in \mathbb{N}$ the maximal depth of the hypertries that are to be stored.*

We denote the set of hypertries $\bigcup_{t \leq d} H(t, A, E)$ as Λ_0. Λ_0 without empty hypertries $\{h \in \Lambda_0 \mid z(h) \neq 0\}$ is denoted Λ.

A hypertrie context *C for hypertries from Λ_0 is defined by a triple (P, m, r) where*

- *P is a bag of elements from Λ_0,*
- *$m : \mathbb{Z} \twoheadrightarrow \Lambda$ maps hashes to non-empty hypertries which are P or are sub-hypertries of one of P's elements, and*
- *$r : \Lambda \to \mathbb{N} \cup 0$ assigns a reference count to non-empty hypertries.*

We define two relations between hypertrie context and hypertries:

- *Hypertries $p \in P$ are primarily contained in C, denoted as $p \overline{\in} C$.*
- *Hypertries $h \in \mathrm{cod}(m)$ are contained in C, denoted as $h \in C$.*

For a hypertrie $h \in \Lambda$, $r(h)$ is calculated from sum of the count of h in P and the number of references to h from hypertrie $h' \in C$:

$$r(h) := \mathit{count}(h, P) + \sum_{\substack{h' \in C \\ p \in \mathbb{I}_d}} \mathit{count}(h, \mathrm{cod}(c_p^{(h')})). \tag{3}$$

[2] The outgoing edges $c_p^{(h)}$ in Definition 1 are considered to map hashes of hypertries instead of hypertries.

4.2 Single-Entry Node

Central properties of a hypertrie are that slicing in any dimension can be carried out efficiently (see R2 and R3 in Sect. 3.4) and that non-zero slices can be iterated efficiently (see R4 in Sect. 3.4). In the implementation of hypertrie node described so far (in the following: full node), this is achieved by maintaining one hash table of non-zero slices for each dimension. The main observation behind this optimization is that R2–R4 also hold for a hypertrie node that represents only a single entry if the hypertrie node stores only the entry itself. We dub such a node *single-entry node* (SEN). A similar technique is used in radix trees [12] to store non-branching paths in a condensed fashion.

For slicing, it is sufficient to match the slice key against the single entry of the node. Thus, the result may have zero or one non-zero entry (see R2, R3). There is exactly one non-zero slice in each dimension. Iteration of the non-zero slices is now trivial (see R4).

SEN are—when applicable—always more memory efficient than full nodes.[3] Compared to a full node h, an SEN eliminates memory overhead in three ways. (1) It does not maintain hash tables $c_p^{(h)}$ for edges to child nodes. (2) Child nodes do not need to be stored, unless they are also needed by other nodes. (3) The node size $z(h)$ does not need to be stored explicitly since it is always 1.

Formally, we define an SEN as follows:

Definition 3 (Single-Entry Hypertrie). *Let H, d, A and E be given as in Definition 1. Further, consider $h \in H(d)$, which stores for key $\langle \mathbf{k}_1, \ldots, \mathbf{k}_n \rangle$, the value v. If h encodes exactly one entry ($z(h) = 1$), h is defined as $\langle \langle \mathbf{k}_1, \ldots, \mathbf{k}_n \rangle, v \rangle$ and is called a* single-entry node *(SEN). Children mapping functions $c_p^{(h)}$ are not defined for h.*

SEN can be used without limitations in a hypertrie context.

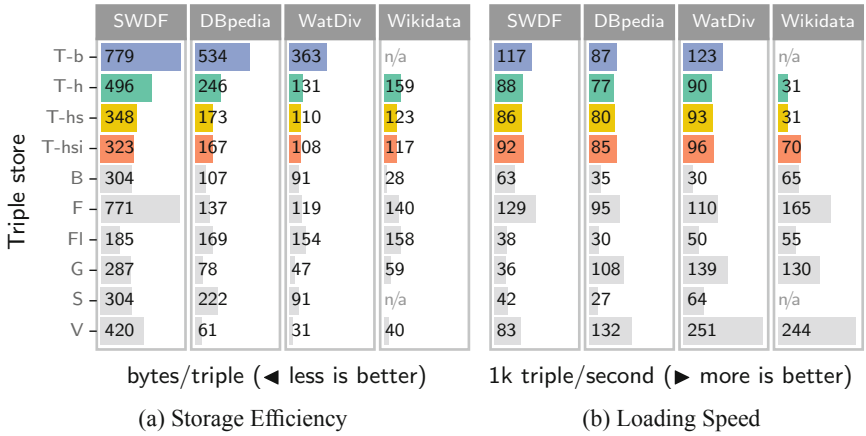
4.3 In-Place Storage

Our third optimization is to store certain nodes exactly where a reference to them would be stored otherwise. While the aforementioned optimizations can be used for hypertries with all value types (e.g., Boolean, integer, float), the optimization in this section is only applicable to Boolean-valued hypertries.

The payload of a binary-valued (note that our tensors only contain 0 s and 1 s) height-1 SEN is a single key part (1 word). It takes the same amount of memory as the hash that identifies the hypertrie (1 word) and which is stored in its parent nodes' children mappings to reference it. Therefore, the payload of a height-1 SEN fits into the place of its reference.

We use this property to reduce the total storage required: The payload of child height-1 SENs—their key part—is stored in place of their reference in the children mappings of their parent nodes. To encode if a hash or a key part is stored, a bit in the

[3] Consider a hypertrie h with a single entry $z(h) = 1$ and depth $d \geq 1$. A hash table that maps a key part requires more memory than just a single key part. Hence, the d child mapping hash tables of a full hypertrie node encoding h require more memory than the d key parts of the entry stored in an SEN encoding h. Consequently, an SEN node is always more memory efficient than a full node.



(a) Storage Efficiency (b) Loading Speed

Fig. 4. Storage efficiency and loading speed of TENTRIS, with different hypertrie versions, and of other triple stores on four datasets. If loading a dataset with a triple store failed, the plot says n/a. The triple stores are TENTRIS (T-*) where * indicates the hypertrie version (see Fig. 1), Blazegraph (B), Fuseki (F), Fuseki-LTJ (Fl), GraphDB (G), gStore (S), and Virtuoso (V).

same fixed position of both key part and hash is reserved and used as a type tagging bit, e.g., the most significant bit. As in-place stored height-1 SEN are not heap-allocated, reference counting is not necessary. The memory is released properly when the hash table is destructed.

4.4 Example

An exemplary comparison of a baseline hypertrie and a hypertrie context containing one primary hypertrie with all three proposed optimizations is given in Fig. 3.

5 Evaluation

We implemented our optimizations within the TENTRIS framework. The goal of our evaluation was twofold: first, we assessed the index sizes and index generation times with four datasets of up to 5.5 B triples. In a second experiment, we evaluated the query performance of the triple stores in a stress test. Throughout our evaluation, we compared the original version of TENTRIS, dubbed TENTRIS-b, our extension of TENTRIS with hash identifiers (h) and single entry nodes (s), dubbed TENTRIS-hs, TENTRIS-hs extended with the in-place storage (i) optimization, dubbed TENTRIS-hsi, and the six popular triple stores, i.e., Blazegraph 2.1.6 Release Candidate, Fuseki 4.4.0, Fuseki-LTJ—a Fuseki that uses a worst-case optimal join algorithm—[4], GraphDB 9.5.1, gStore 0.8[5], and Virtuoso 7.2.6.1. We chose popular triple stores which provide a standard

[4] Fuseki-LTJ is based on Apache Jena Fuseki 3.9.0.

[5] We used the modified version from [6] that fixes the SPARQL endpoint and sets a query timeout.

HTTP SPARQL interface, support at least the same subset of SPARQL as TENTRIS and are freely available for benchmarking. We did not include Qdag or Ring because they do not provide a SPARQL HTTP endpoint and do not support projections. We used the datasets Semantic Web Dog Food (SWDF) (372 K triples), the English DBpedia version 2015-10 (681 M triples) and WatDiv [2] (1 B triples) and their respective query lists from [6]. We added Wikidata trusty from 2020-11-11 (5.5 B triples) as another large real-world dataset and generated queries with FEASIBLE [19] from Wikidata query logs. As in [6], FEASIBLE was configured to generate `SELECT` queries with BGPs and `DISTINCT` as an optional solution modifier. All experiments were executed on a server with an AMD EPYC 7742, 1 TB RAM and two 3 TB NVMe SSDs in RAID 0 running Debian 10 and OpenJDK 11.0.14.

5.1 Index Size and Loading Time

Storage requirements for indices and index building speeds are reported in Fig. 4. The index sizes of the TENTRIS versions were measured with `cgmemtime`'s[6]"Recursive and acc. high-water RSS+CACHE". For all other triple stores, the total size of the index files after loading was used. `cgmemtime`'s "Child wall" was used to measure the time for loading the datasets.

Two triple stores were not able to load the Wikidata dataset: gStore failed due to a limit on the number of usable RDF Resources and TENTRIS-b ran out of memory.

For all datasets, each additional hypertrie optimization improves the storage efficiency of TENTRIS further: Compared to TENTRIS-b, the optimizations h, hs and hsi take 36–64%, 55–68% and 58–70% less memory respectively. This comes at the cost of decreased index build throughput for TENTRIS-h and TENTRIS-hs by 11–36% and for TENTRIS-his by 2–28%. For the Wikidata dataset, the index sizes of TENTRIS-h, TENTRIS-hs and TENTRIS-hsi are reduced by at least 21%, 39% and 42%[7], respectively. Compared to TENTRIS-h, the single entry nodes (s) in TENTRIS-hs save 16–30% with almost no effect on the index building speed. The in-place storage of single-entry leaf nodes (i) in TENTRIS-hsi saves memory, (another 1–7%) compared to TENTRIS-hs, and speeds up the index building (2–57%) on all datasets. For the small to medium-sized datasets SWDF, DBpedia, and WatDiv, the index building is slightly faster by 2–7%; for the large dataset, Wikidata, the margin is considerably larger with 56% improvement.

The index sizes of all TENTRIS versions scale similarly to other triple stores. The TENTRIS-hsi indices are similar in size to the indices produced by other triple stores. Compared to the smallest index for each dataset, TENTRIS-hsi uses 1.14 to 4.24 times more space. The loading time of TENTRIS-hsi is close to the mean of the non-TENTRIS triple stores.

[6] https://github.com/gsauthof/cgmemtime.

[7] In comparison to 1TB RAM because TENTRIS-b ran out memory during loading.

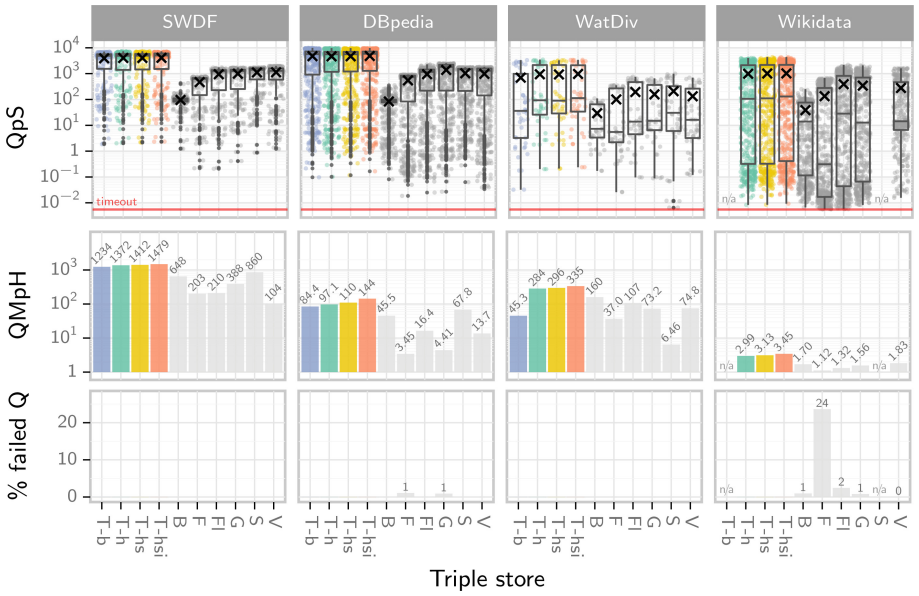


Fig. 5. Performance metrics of query stress tests on four benchmarks for TENTRIS, with different hypertrie versions, and for other triple stores. For triple store abbreviations see Fig. 4. First row shows Boxplots and scatterplots for Queries per Second (QpS). Only successful query executions are considered and aggregated by mean into a single scatter. The boxes indicate the first quartile, median, and third quartile with 1.5 times the interquartile range wiskers. Black dots mark outliers and crosses the means. Second row shows shows Query Mixes per Hour (QMpH). Failed queries are rated with the timeout duration of **180 s**. The third row shows the percentage of queries that failed. If no number is provided, no queries failed. If experiments were not executed for a combination of triple store and benchmark, the plot says n/a.

5.2 Querying Stress Test

Our evaluation setup for query stress tests was similar to that used in [6]. The results are shown in Fig. 5. The experiments were executed using the benchmark execution framework IGUANA v3.2.1 [7]. For each benchmark, the query mix was executed 30 times on each triple store and the timeout for a single query execution was set to 3 min. We report the performance using Queries per Second (QpS), Query Mixes per Hour (QMpH) and the proportion of failed queries. For QpS, only query executions that were successful and finished before the timeout are considered. The reported QpS value of a query on a dataset and triple store is the mean of the single measurements. Failed queries are penalized with the timeout duration for QMpH. We chose to report both QMpH and QpS to get a more fine-grained view of the performance. While QpS is more robust against outliers, QMpH can be strongly influenced by long-running and failed queries.

The baseline version of TENTRIS, TENTRIS-b, already performs better than all non-TENTRIS triple stores w.r.t. QpS and QMpH on the SWDF and DBpedia benchmarks but not on the WatDiv benchmark. Here, gStore and Blazegraph outperform TENTRIS-b by a factor of 1.6 and 3.5 with respect to QMpH. TENTRIS-h, TENTRIS-hs and TENTRIS-hsi all outperform TENTRIS-b and all other triple stores on all datasets with respect to average QpS (avgQpS) and QMpH. For the small real-world dataset SWDF, all TENTRIS versions answered queries with similar avgQpS ranging from 3935 (TENTRIS-b) to 4088 (TENTRIS-hsi, +4%). The same holds true for the larger real-world dataset DBpedia, with avgQpS ranging from 4753 (TENTRIS-b) to 4825 (TENTRIS-hsi, +1.5%). With respect to QMpH, the optimized TENTRIS versions clearly outperform TENTRIS-b by 11%(TENTRIS-h), 14% (TENTRIS-hs) and 20% (TENTRIS-hsi) on the SWDF dataset, and even by 15%, 31% and 71% on the DBpedia dataset. On the synthetic dataset WatDiv, the optimized TENTRIS versions show notable speedups on both metrics, avgQpS and QMpH. AvgQpS is increased from 698 by TENTRIS-h to 946 (+35%), by TENTRIS-hs, to 931 (+33%), by TENTRIS-hs and to 972 (+39%) by TENTRIS-hsi. QMpH is 5.2, 6.6 and 7.4 times higher with TENTRIS-h, TENTRIS-hs and TENTRIS-hsi, respectively, than with TENTRIS-b.

On Wikidata, measurements are available only for the TENTRIS versions h, hs and hsi due to TENTRIS-b not being able to load the dataset. TENTRIS-hsi is again slightly faster than TENTRIS-hs, with 1009 (hs) and 1021 (+1%, hsi) avgQpS, and 3.13 (hs) and 3.45 (+9%, hsi) QMpH. TENTRIS-h is with 989 avgQpS and 2.99 QMpH slightly slower than the more optimized versions.

When compared to the fastest non-TENTRIS triple store on each metric and dataset, TENTRIS-hsi is 3–3.7 times faster with respect to avgQpS and 1.7–2.1 times faster with respect to QMpH. None of the TENTRIS versions had failed queries during execution. On the DBpedia dataset, Fuseki and gStore failed on about 1% of the queries. On the Wikidata dataset, all non-TENTRIS triples stores that succeeded to load the dataset failed on some queries.

5.3 Discussion

The evaluation shows that applying all three optimizations (hsi) is in all aspects superior to applying only the first two optimizations (h, hs). Thus, we will consider only TENTRIS-hsi in the following. The proposed optimizations of the hypertrie improve the storage efficiency by 70% and the query performance with respect to avgQpS by large margins of up to four orders of magnitude. These improvements come at the cost of slightly longer index building times of at most 28%. The optimization of the storage efficiency is clearly attributable to the reduced number of nodes, as shown in Fig. 1. For the improved query performance, definite attribution is difficult. We worked out two main factors we believe are reasonable to assume as the cause: First, information that was stored in a node and its subnodes in the baseline version is in the optimized version more often stored in a single node. This way, the optimizations single-entry node (s) and in-place storage (i) cause fewer CPU cache misses and fewer resolves of memory addresses, resulting in faster execution. Second, key parts are not necessarily stored in a hash table anymore. Whenever a key part is read from a single-entry node (s) or in-place stored node (i), the optimized version saves one hash table lookup compared

to the baseline version. On the other side, additional hash table lookups are required to retrieve nodes by their hash identifiers during query evaluation. We minimize this overhead by handling nodes by their memory address during evaluation after they were looked up by their hash first. The memory overhead for storing these handles is negligible as typically only a few are required at the same time.

For triple stores, there is always a trade-off between storage efficiency, index build time, and query performance. In particular, less compressed indices can typically be built faster. Building multiple indices takes longer but multiple indices allow for more optimized query plans. The baseline hypertrie clearly attributed significant weight to good query performance, with average index building time and above average storage requirement. The optimized hypertrie trades a slightly little worse index building time for better query performance and much-improved storage efficiency. The result is a triple store with superior query performance, average storage requirement, and still average index building time. Given the predominantly positive changes in trade-offs, we consider the proposed optimizations a substantial improvement.

6 Conclusion and Outlook

We presented a memory-optimized version of the hypertrie data structure. The three optimizations of hypertries that we developed and evaluated improved both the memory footprint and query performance of hypertries. A clear but small trade-off of our approaches is the slightly longer index building time they require.

The new storage scheme for hypertrie opens up several new avenues for future improvements. The persistence of optimized hypertrie nodes is easier to achieve due to the switch from memory pointers to hashes. Furthermore, the hash identifiable hypertrie nodes provide the building bricks to distribute a hypertrie over multiple nodes in a network. For TENTRIS, the introduction of the hypertrie context opens up the possibility to store the hypertries of multiple RDF graphs in a single context and thereby automatically deduplicate common sub-hypertries. Especially for similar graphs, this optimization has the potential to improve storage efficiency substantially.

Supplementary Material Statement: Source code for our system; a script to recreate the full experimental setup, including all datasets, queries, triple stores, configurations and scripts to run the experiments; and the raw data and scripts for generating the images are available from: https://tentris.dice-research.org/iswc2022.

Acknowledgments. The authors would like to thank Lukas Kerkemeier for his work on the implementation. This work has been supported by the German Federal Ministry for Economic Affairs and Climate Action (BMWK) within the project RAKI under the grant no 01MD19012B, by the German Federal Ministry of Education and Research (BMBF) within the EuroStars project E!114681 3DFed under the grant no 01QE2114B and by the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 860801.

References

1. Ali, W., Saleem, M., Yao, B., Hogan, A., Ngomo, A.C.N.: A survey of rdf stores & sparql engines for querying knowledge graphs (2021)
2. Aluç, G., Hartig, O., Özsu, M.T., Daudjee, K.: Diversified stress testing of RDF data management systems. In: Mika, P., et al. (eds.) ISWC 2014. LNCS, vol. 8796, pp. 197–212. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11964-9_13
3. Arroyuelo, D., Hogan, A., Navarro, G., Reutter, J.L., Rojas-Ledesma, J., Soto, A.: Worst-case optimal graph joins in almost no space, pp. 102–114. Association for Computing Machinery, New York (2021). https://doi.org/10.1145/3448016.3457256
4. Atre, M., Chaoji, V., Zaki, M.J., Hendler, J.A.: Matrix "bit" loaded: a scalable lightweight join query processor for RDF data. In: Proceedings of the 19th International Conference on World Wide Web, WWW 2010, pp. 41–50. Association for Computing Machinery, New York (2010). https://doi.org/10.1145/1772690.1772696
5. Atserias, A., Grohe, M., Marx, D.: Size bounds and query plans for relational joins. In: 49th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2008, Philadelphia, PA, USA, 25–28 October 2008, pp. 739–748. IEEE Computer Society (2008). https://doi.org/10.1109/FOCS.2008.43
6. Bigerl, A., Conrads, F., Behning, C., Sherif, M.A., Saleem, M., Ngonga Ngomo, A.-C.: Tentris – a tensor-based triple store. In: Pan, J.Z., et al. (eds.) ISWC 2020. LNCS, vol. 12506, pp. 56–73. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-62419-4_4
7. Conrads, F., Lehmann, J., Saleem, M., Morsey, M., Ngonga Ngomo, A.-C.: IGUANA: a generic framework for benchmarking the read-write performance of triple stores. In: d'Amato, C., et al. (eds.) ISWC 2017. LNCS, vol. 10588, pp. 48–65. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68204-4_5
8. Einstein, A.: Die Grundlage der allgemeinen Relativitätstheorie. Annalen der Physik **354**, 769–822 (1916). https://doi.org/10.1002/andp.19163540702
9. Erling, O.: Virtuoso, a hybrid RDBMS/graph column store. http://vos.openlinksw.com/owiki/wiki/VOS/VOSArticleVirtuosoAHybridRDBMSGraphColumnStore. Accessed 17 Mar 2018
10. Foundation, A.S.: Apache jena documentation - TDB architecture (2019). https://jena.apache.org/documentation/tdb/architecture. Accessed 25 Apr 2019
11. Hogan, A., Riveros, C., Rojas, C., Soto, A.: A worst-case optimal join algorithm for SPARQL. In: Ghidini, C., et al. (eds.) ISWC 2019. LNCS, vol. 11778, pp. 258–275. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30793-6_15
12. Leis, V., Kemper, A., Neumann, T.: The adaptive radix tree: artful indexing for main-memory databases. In: 2013 IEEE 29th International Conference on Data Engineering (ICDE), pp. 38–49 (2013). https://doi.org/10.1109/ICDE.2013.6544812
13. Malhotra, J., Bakal, J.: A survey and comparative study of data deduplication techniques. In: 2015 International Conference on Pervasive Computing (ICPC), pp. 1–5 (2015). https://doi.org/10.1109/PERVASIVE.2015.7087116
14. Morrison, D.R.: Patricia-practical algorithm to retrieve information coded in alphanumeric. J. ACM **15**(4), 514–534 (1968). https://doi.org/10.1145/321479.321481
15. Navarro, G., Reutter, J.L., Rojas-Ledesma, J.: Optimal joins using compact data structures. In: 23rd International Conference on Database Theory, ICDT 2020, March 30-April 2, 2020, Copenhagen, Denmark. LIPIcs, vol. 155, pp. 21:1–21:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020). https://doi.org/10.4230/LIPIcs.ICDT.2020.21, https://doi.org/10.4230/LIPIcs.ICDT.2020.21
16. Neumann, T., Weikum, G.: Rdf-3x: a risc-style engine for RDF. In: Proceedings VLDB Endowment, vol. 1, no. 1, pp. 647–659 (2008). https://doi.org/10.14778/1453856.1453927

17. Ontotext USA, I.: Storage - GraphDB free 8.9 documentation. http://graphdb.ontotext.com/documentation/free/storage.html#storage-literal-index. Accessed 16 Apr 2019
18. Ricci, M., Levi-Civita, T.: Méthodes de calcul différentiel absolu et leurs applications. Mathematische Annalen **54**(1–2), 125–201 (1900)
19. Saleem, M., Mehmood, Q., Ngonga Ngomo, A.-C.: FEASIBLE: a feature-based SPARQL benchmark generation framework. In: Arenas, M., et al. (eds.) ISWC 2015. LNCS, vol. 9366, pp. 52–69. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25007-6_4
20. SYSTAP, LLC: Bigdata Database Architecture - Blazegraph (2013). https://blazegraph.com/docs/bigdata_architecture_whitepaper.pdf. Accessed 29 Nov 2019
21. Veldhuizen, T.L.: Triejoin: a simple, worst-case optimal join algorithm. In: Proceedings of 17th International Conference on Database Theory (ICDT), Athens, Greece, 24–28 March 2014, pp. 96–106. OpenProceedings.org (2014). https://doi.org/10.5441/002/icdt.2014.13
22. Zou, L., Özsu, M.T., Chen, L., Shen, X., Huang, R., Zhao, D.: Gstore: a graph-based sparql query engine. VLDB J. **23**(4), 565–590 (2014). https://doi.org/10.1007/s00778-013-0337-7