Tentris – A Tensor-Based Triple Store

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

[1] DICE Group, CS Department, Paderborn University, Paderborn, Germany
{alexander.bigerl,felix.conrads,mohamed.sherif}@uni-paderborn.de, 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, University of Leipzig, Leipzig, Germany
saleem@informatik.uni-leipzig.de
https://dice-research.org/

Abstract. The number and size of RDF knowledge graphs grows continuously. Efficient storage solutions for these graphs are indispensable for their use in real applications. We present such a storage solution dubbed TENTRIS. Our solution represents RDF knowledge graphs as sparse order-3 tensors using a novel data structure, which we dub hypertrie. It then uses tensor algebra to carry out SPARQL queries by mapping SPARQL operations to Einstein summation. By being able to compute Einstein summations efficiently, TENTRIS outperforms the commercial and open-source RDF storage solutions evaluated in our experiments by at least 1.8 times with respect to the average number of queries it can serve per second on three datasets of up to 1 billion triples. Our code, evaluation setup, results, supplementary material and the datasets are provided at https://tentris.dice-research.org/iswc2020.

1 Introduction

A constantly increasing amount of data is published as knowledge graphs. Over 149 billion facts are published in the 2973 datasets of the Linked Open Data (LOD) Cloud [9], including large datasets such as UniProt[1] (55.3 billion facts) and LinkedTCGA [22] (20.4 billion facts). Even larger knowledge graphs are available in multinational organisations, including Google, Microsoft, Uber and LinkedIn [18]. Proposing scalable solutions for storing and querying such massive amount of data is of central importance for the further uptake of knowledge graphs. This has motivated the development of a large number of solutions for their storage and querying [4,8,10,12,15,19,24,27–29].

We present TENTRIS, a new in-memorytriple store for the efficient storage and querying of RDF data. Two innovations are at the core of our triple store. First, TENTRIS represents RDF data as sparse order-3 tensors using a novel in-memorytensor data structure dubbed *hypertrie*, which we also introduce in this paper. This data structure facilitates the representation of SPARQL queries as (sequences of) operations on tensors. A hypertrie combines multiple indexes into a single data structure, thus eliminating

[1] https://www.uniprot.org/downloads.

J. Z. Pan et al. (Eds.): ISWC 2020, LNCS 12506, pp. 56–73, 2020.
https://doi.org/10.1007/978-3-030-62419-4_4

some of the redundancies of solutions based on multiple indexes (see, e.g., [15]). As a result, TENTRIS can store whole knowledge graphs and corresponding indexes into a single unified data structure.

Our second innovation lies in the way we process SPARQL[2] queries: To query the RDF data stored in TENTRIS, SPARQL queries are mapped to Einstein Summations. As a result, query optimization is delegated to the implementation of an Einstein summation operator for hypertries. Since the proposed tensor data structure offers precise statistics, the order for tensor operations is computed online, thus further speeding up the query execution.

The rest of the paper is organized as follows: Sect. 2 gives an overview of related work. In Sect. 3 notations are defined and backgrounds on tensors are provided. In Sect. 4 the mapping of RDF graphs to tensors is defined and in Sect. 5 we introduce our new tensor data structure. In Sect. 6 the querying approach is described. The evaluation is presented in Sect. 7, and in Sect. 8 we conclude and look at future prospects. Examples for definitions and concepts are provided throughout the paper. For an extended, comprehensive example, see the supplementary material.

2 Related Work

Several commercial and open-source triple stores have been developed over recent years and used in a number of applications. In the following, we briefly introduce the most commonly used triple stores that are documented and freely available for benchmarking. We focus on these triple stores because they are candidates for comparison with our approach. We do not consider distributed solutions (see [1] for an overview), as a distributed version of TENTRIS will be the subject of future work. Throughout our presentation of these approaches, we sketch the type of indexes they use for storing and querying RDF, as this is one of the key differences across triple stores.[3]

RDF-3X [15] makes extensive use of indexes. This triple store builds indexes for all full collation orders SPO (Subject, Predicate, Object), SOP, OSP, OPS, PSO, POS, all aggregated indexes SP, PS, SO, OS, PO, OP and all one-value indexes S, P and O. It uses a B^+-tree as index data structure that is extended by LZ77 compression to reduce the memory footprint. Virtuoso [8] uses "2 full indexes over RDF quads plus 3 partial indexes" [8], i.e., PSOG (Predicate, Subject, Object, Graph), POGS, SP, OP and GS. Apache Jena TDB2 [10] uses three indexes to store the triples in the collation orders SPO, POS, and OSP. The indexes are loaded via memory mapped files. GraphDB [19] uses PSO and POS indexes to store RDF statements. BlazeGraph [24] uses B^+-trees as data structure for its indexes. Statements are stored in the collation orders SPO,

[2] At the moment TENTRIS supports the same fragment of SPARQL as [4,12,26,27] which includes basic graph patterns and projections with or without DISTINCT.

[3] Note that indexes for different collation orders are crucial for the performance of triple stores. They determine which join orders are possible and which triple patterns are cheap to resolve. However, building indexes comes at a cost: each index takes additional time to build and update. It also requires additional memory. Consequently, there is always a trade-off between querying speed on the one hand and memory consumption and maintenance cost on the other hand.

POS, and OSP. gStore [29] uses a signature-based approach to store RDF graphs and to answer SPARQL queries. An RDF graph is stored in an extended signature tree, called VS*-tree. Additionally, it generates materialized views to speed up star queries. In contrast to most other triple stores, gStore derives signatures from RDF terms instead of using unique IDs. gStore is a in-memory system, i.e., it holds all data in main memory.

Another in-memory triple store, RDFox [14], uses a triple table with three additional rows which store linked lists of triples with equal subjects, predicates and object respectively. The elements of the subjects and objects lists are grouped by predicates. Indices on the triple table are maintained for collation orders S, P, and O as arrays, and for collation orders SP, OP and SPO as hashtables.

The idea of using matrices or tensors to build triple stores has been described in a few publications. BitMat [4], like TENTRIS, uses an order-3 Boolean tensor as an abstract data structure for an RDF graph. The actual implementation stores the data in collation orders PSO and POS. The subindexes for SO and OS are stored using Boolean matrices. Join processing is done using a multi-way join approach. However, BitMat is unable to answer queries that use variables for predicates in triple patterns, i.e., `SELECT ?p WHERE {a ?p b.}`. A similar approach was chosen by the authors of TripleBit [28]. For each predicate, this approach stores an SO and OS index based on a custom column-compressed matrix implementation. This results in two full indexes—PSO and POS. In contrast to BitMat, TripleBit supports variables for predicates in triple patterns. A more generic approach is used for MagiQ [12]. The authors define a mapping of RDF and SPARQL to algebraic structures and operations that may be implemented with different linear algebra libraries as a backend. The RDF graph is encoded into a sparse matrix. A statement is represented by using predicates as values and interpreting the column and row number as subject and object IDs. Basic graph patterns are translated to general linear algebra operations. The approach does not support variables for predicates in triple patterns. A similar mapping was also chosen by the authors of TensorRDF [27] using Mathematica as a backend for executing matrix operations. All mentioned triple stores except gStore use unique IDs to represent each resource. They store the mapping in an index for query translation and result serialization. Further, all of them except gStore apply column-oriented storage.

Like most stores above, TENTRIS adopts the usage of unique IDs for resources and column-oriented storage. However, it does not use multiple independent indexes or materialized views. Instead, TENTRIS relies on the novel hypertrie tensor data structure that unifies multiple indexes into a single data structure. Like gStore and RDFox, it holds all data in-memory. In contrast to some of the other tensor-based solutions, TENTRIS can process queries which contain unbound predicates.

3 Background

3.1 Notation and Conventions

Let $\mathbb{B}$ be the set of Boolean values, i.e., $\{\mathit{true}, \mathit{false}\}$ and $\mathbb{N}$ be the set of the natural numbers including 0. We map *true* to 1 and *false* to 0. The natural numbers from 1 to n are shorthanded by $\mathbb{I}_n := \{i \in \mathbb{N} \mid 1 \leq i \leq n\}$. The set of functions $\{f \mid f : X \to Y\}$ is denoted Y^X or $[X \to Y]$. The domain of a function f is written as $\operatorname{dom}(f)$ and the target (also called codomain) is denoted by $\operatorname{cod}(f)$. 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 like $l = \langle a, b, c\rangle$. Their elements are accessible via subscript, e.g., $l_1 = a$. Bags are delimited by curly-pipe brackets, e.g., $\{\!| a, a, b |\!\}$. The number of times an element e is contained in any bag or sequence C is denoted by $\mathit{count}(e, C)$; for example, $\mathit{count}(a, \{\!| a, a, b |\!\}) = 2$. We denote the Cartesian product of S with itself i times with $S^i = \underbrace{S \times S \times \ldots S}_{i}$.

3.2 Tensors and Tensor Operations

In this paper, we limit ourselves to tensors that can be represented as finite n-dimensional arrays.[4] An order-n tensor T is defined as a mapping from a finite multi-index $\mathbf{K} = \mathbf{K}_1 \times \cdots \times \mathbf{K}_n$ to some codomain V. We only use multi-indexes with $\mathbf{K}_1 = \cdots = \mathbf{K}_n \subset \mathbb{N}$. In addition, we consider exclusively tensors T with $\mathbb{B}$ or $\mathbb{N}$ as codomain. We call $\mathbf{k} \in \mathbf{K}$ a key with key parts $\langle \mathbf{k}_1, \ldots, \mathbf{k}_n\rangle$. Values v in a tensor are accessed in array style, e.g., $T[\mathbf{k}] = v$.

Example 1. An example of an order-3 tensor $T \in [(\mathbb{I}_8)^3 \to \mathbb{B}]$ is given in Fig. 1. Only those entries given by the points in the figure are set to 1.

Slices. Slicing is an operation on a tensor T that returns a well-defined portion of T in the form of a lower-order tensor. Slicing is done by means of a slice key $\mathrm{s} \in \mathbf{S} := \mathbf{K}_1 \cup \{:\} \times \cdots \times \mathbf{K}_n \cup \{:\}$ with: $\notin \mathbf{K}_1, \ldots, \mathbf{K}_n$. When applying s to a tensor T (denoted $T[\mathrm{s}]$), the dimensions marked with : are kept. A slice key part $\mathrm{s}_i \neq$: removes all entries with other key parts at position i and removes $\mathbf{K}_i$ from the result's domain. The sequence brackets may be omitted from the notation, e.g., $T[:, 2, :]$ for $T[\langle :, 2, :\rangle]$.

Example 2. Let T be the tensor from Example 1. The slice $T[\mathrm{s}]$ with the slice key $\mathrm{s} = \langle :, 2, :\rangle$ is an order-2 tensor with 1 at keys $\langle 1, 3\rangle$, $\langle 1, 4\rangle$, $\langle 3, 4\rangle$, $\langle 3, 5\rangle$, $\langle 4, 3\rangle$ and $\langle 4, 5\rangle$.

Definition 1. *Assume T, $\mathbf{K}$, V, n, $\mathbf{S}$ and s to be defined as above. Let P be the sequence of positions in s which are set to :. For $\mathrm{s} = \langle :, 2, :\rangle$, P would be $\langle 1, 3\rangle$. A sub-multi-index is defined by $\mathbf{K}' := \bigtimes_{i \in P} \mathbf{K}_i$. Keys from the sub-multi-index are mapped to the original multi-index by $\varphi_{\mathrm{s}} : \mathbf{K}' \to \mathbf{K}$ with*

$$\varphi_{\mathrm{s}} : \mathbf{k}' \mapsto \mathbf{k} \textit{ with } \mathbf{k}_i = \begin{cases} \mathbf{k}'_j & \textit{if } i = P_j, \\ \mathrm{s}_i & \textit{otherwise.} \end{cases}$$

A slice $T' = T[\mathrm{s}]$ can now be defined formally as follows: $T' \in V^{\mathbf{K}'} : \mathbf{k}' \mapsto T[\varphi_{\mathrm{s}}(\mathbf{k}')]$.

[4] Tensors can be defined in a more general manner than provided herein, see [2] for details.

Table 1. Example RDF Graph. Resources are printed alongside their integer IDs. The integer IDs are enclosed in brackets and are not part of the resource.

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)

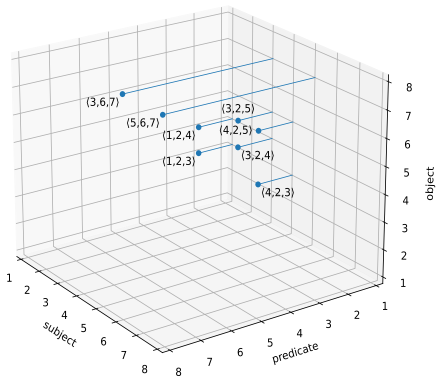

Fig. 1. 3D plot of the tensor that represents the RDF graph from Table 1. Every 1 is indicated by a point at the corresponding position. Points are orthogonally connected to the subject-object-plane for better readability.

Einstein Summation. We define Einstein summation in a manner akin to [13]. Einstein summation is a variable-input operation that makes the combination of multiple operations on vectors, matrices and tensors in a single expression possible [7,20]. Einstein summation is available in many modern tensor and machine learning frameworks [11,13,25,26]. It supports, amongst others, inner products, outer products, contractions and scalar multiplications. "The notation uses [subscript labels] to relate each [dimension] in the result to the [dimension] in the operands that are combined to produce its value." [13, p. 77:3]

Example 3. Consider the tensor T from Example 2 and slices $T^{(1)} := T[1,2,:]$, $T^{(2)}[:,2,:]$ and $T^{(3)} := T[:,6,7]$. An exemplary Einstein summation is given by $R_f \leftarrow T^{(1)}_f \times T^{(2)}_{f,u} \times T^{(3)}_u$. The result R is an order-1 tensor, which is calculated as $R[f \in \mathbb{I}_8] = \sum_{u \in \mathbb{I}_8} T^{(1)}[f] \cdot T^{(2)}[f,u] \cdot T^{(3)}[u]$, and results in $R = \langle \underset{1}{0}, \underset{2}{0}, \underset{3}{1}, \underset{4}{2}, \underset{5}{0}, \underset{6}{0}, \underset{7}{0}, \underset{8}{0} \rangle$.

We use Einstein notation expressions on the semiring $(\mathbb{N}, +, 0, \cdot, 1)$ to support bag semantics for SPARQL results. We also implement set semantics for `DISTINCT` queries using $(\mathbb{B}, \vee, 0, \wedge, 1)$ as semiring. All corresponding definitions are analogous to those presented in the paper for bag semantics and are hence not detailed any further.

4 RDF Graphs as Tensors

Our mapping of RDF graphs to order-3 tensors extends the model presented in [17] by adding a supplementary index, which serves to map undefined variables in SPARQL solution mappings. By adopting the same representation for RDF graphs and bags of solution mappings, we ensure that graphs and bags of mappings are compatible and can hence conjoint in tensor operations. Informally, the tensor $T(g)$ of an RDF graph g with α resources is hence an element of $[(\mathbb{I}_{\alpha+1})^3 \rightarrow \mathbb{B}]$ such that $T[i,j,k] = 1$ holds iff the i-th resource of g is connected to the k-th resource of g via the j-th resource (which must be a predicate) of the same graph. Otherwise, $T[i,j,k] = 0$.

Example 4. Consider the triples of the RDF graph g' shown in Table 1. Each RDF term of g' is printed alongside an integer identifier that is unique to each term. All entries shown in Fig. 1 are set to 1. All other entries are 0.

Formally, we define the tensor $T(g)$ for an RDF graph g:

Definition 2. *Let g be an RDF graph and $r(g)$ the set of RDF terms used in g. We define id as a fixed bijection that maps RDF terms $r(g)$ and ϵ, a placeholder for undefined variables in SPARQL solution mapping, to integer identifiers $\mathbb{I} := \mathbb{I}_{|r(g)|+1}$. The inverse of id is denoted id^{-1}. With respect to g and id, an RDF term $\langle s,p,o\rangle$ is represented by a key $\langle id(s), id(p), id(o)\rangle$. The tensor representation of g is given by $t(g) \in [\mathbb{I}^3 \rightarrow \mathbb{B}]$. The entries of $t(g)$ map a key $\mathbf{k}$ to 1 if the RDF statement corresponding to $\mathbf{k}$ is in g; otherwise $\mathbf{k}$ is mapped to 0:*

$$\forall \mathbf{k} \in \mathbb{I}^3 : t(g)[\mathbf{k}] := \mathit{count}\,(\langle id^{-1}(\mathbf{k}_1), id^{-1}(\mathbf{k}_2), id^{-1}(\mathbf{k}_3)\rangle, g).$$

The results of a SPARQL query on g is a bag of solution mappings Ω with variables U. Let $\langle u_1, \ldots, u_{|U|}\rangle$ be an arbitrary but fixed sorting of U. A solution mapping $[u_1 \rightarrow w_1, \ldots, u_{|U|} \rightarrow w_{|U|}]$ with $w_i \in r(g) \cup \{\epsilon\}$ is represented by a key $\langle id(w_1), \ldots, id(w_{|U|})\rangle$.[5] The tensor representation of Ω is an order-$|U|$ tensor $t(\Omega)$ where each variable $u \in U$ is mapped to a separate dimension. $t(\Omega)$ maps a key $\mathbf{k}$ to the count of the represented solution mapping in Ω:

$$\forall \mathbf{k} \in \mathbb{I}^{|U|} : t(\Omega)[\mathbf{k}] := \mathit{count}\,([u_1 \rightarrow id^{-1}(\mathbf{k}_1), \ldots, u_{|U|} \rightarrow id^{-1}(\mathbf{k}_{|U|})], \Omega).$$

5 Hypertries – A Data Structure for Tensors

Using tensors for RDF graphs requires a data structure that fulfills the following requirements: (R1) the data structure must be memory-efficient and (R2) must allow efficient slicing (R3) by any combination of dimensions (also see Sect. 6.1). Additionally, (R4) such a data structure must provide an efficient way to iterate the non-zero slices of any of its dimensions.

A trie [6] with a fixed depth is a straightforward sparse tensor representation that fulfills (R1) and (R2). A key consisting of consecutive key parts is stored by adding a path labeled with these key parts from the root node of the trie. Existing labeled edges are reused, introducing a moderate amount of compression (R1). Further, the trie sparsely encodes a Boolean-valued tensor by only storing those keys that map to 1 (R1). Descending by an edge, representing a key part k, is equal to slicing the tensor with the first key part fixed to k. The descending is efficient (R2) if a hashtable or a search tree is used to store the children of a node. However, to support efficient slicing by any other dimension except the first, a new trie with another collation order must be populated. The same holds true for iterating non-zero slices as required for joins (see (R4)).

[5] Technically, SPARQL semantics define solution mappings as partial functions f. Our formal model is equivalent and simply maps all variables for which f is not defined to ϵ.

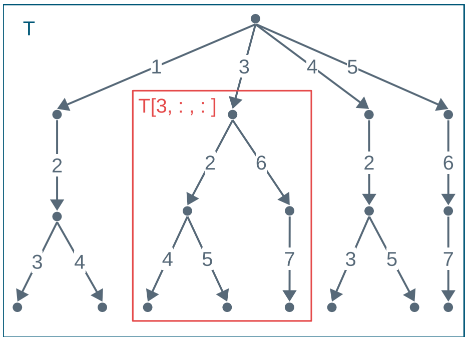

Fig. 2. Trie representation of a tensor $\mathbf{T}$ depicting the data from Table 1. A slice $\mathbf{T}[\mathbf{3},:,:]$ by the first dimension with 3 is shown in the red box. (Color figure online)

Example 5. Figure 2 shows an order-3 Boolean tensor stored in a trie. Each leaf encodes a 1 value for the key that is encoded on the path towards it. The slice for the key $\langle 3,:,:\rangle$ is shown in the red box, resulting in an order-2 tensor.

These limitations are overcome by *hypertries*, a generalization of fixed-depth tries. A hypertrie permits the selection of a key part at an arbitrary position to get a (sub-) hypertrie that holds the concatenations of the corresponding key prefixes and suffixes. To achieve this goal, a node holds not only a set of edges for resolving the first key part, but also a set for every other dimension. This allows for slicing by any dimension as required by condition (R3) above. By storing each dimension's edges and children in a hashmap or search tree, iterating the slices by any dimension is accomplished implicitly. Hence, hypertries fulfill (R4).

Formally, we define a hypertrie as follows:

Definition 3. *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 $c_i^{(h')} : A \nrightarrow H(d-1), i \in \mathbb{I}_n$ 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.*

An example of a hypertrie is given in Fig. 3. A naive implementation of a hypertrie would require as much memory as tries in every collation order. However, we can take advantage of the fact that the slicing order relative to the original hypertrie does not matter when chaining slices. For example, consider a hypertrie h of depth 3. It holds that $h[3,:,:][:,4] = h[:,:,4][3,:]$. Consequently, such equivalent slices should be stored only once and linked otherwise. By applying this technique, the storage bound is reduced from $\mathcal{O}(d! \cdot d \cdot z(h))$ for tries in any collation order to $\mathcal{O}(2^{d-1} \cdot d \cdot z(h))$ for a hypertrie (for proof see supplementary material). Given that d is fixed to 3 for RDF graphs, this

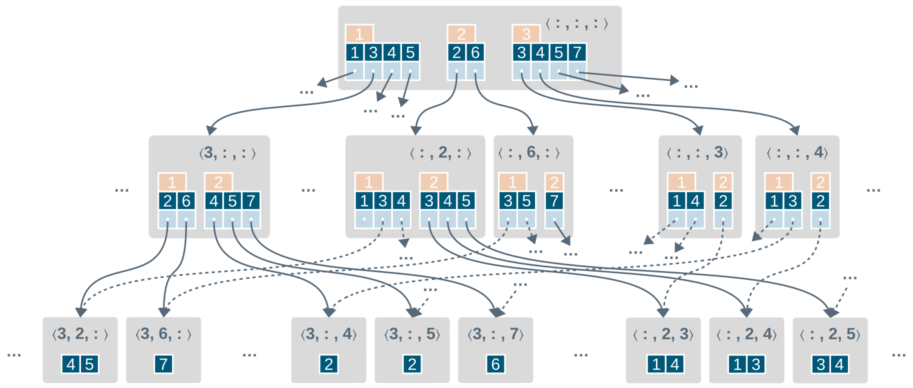

Fig. 3. A hypertrie storing the IDs from Table 1. Most nodes are left out for better readability. Each node represents a non-empty slice of the parent node. The slice key relative to the root node is printed in the node. The orange numbers indicate slice key positions, the mapping below them link all non-empty slices by that position to the nodes encoding the slice result.

results in a data structure that takes at most 4 times more memory than storing the triples in a list. Note that storing all tries for all six collation orders (see, e.g., [15]) requires 6 times as much memory as storing the triples in a list.

6 Querying

6.1 From SPARQL to Tensor Algebra

Triple Pattern. Let g be an RDF graph with the tensor representation T and index function id as defined in Definition 2. Let Q be a triple pattern with variables U and let $Q(g)$ be the bag of solutions that results from applying Q to g. The slice key $\mathbf{k}^{(Q)}$ which serves to execute Q on T is given by

$$\mathbf{k}_i^{(Q)} := \begin{cases} : & \text{if } Q_i \in U, \\ id(Q_i), & \text{otherwise.} \end{cases}$$

If $\mathbf{k}^{(Q)}$ is defined,[6] it holds true that $T[\mathbf{k}^{(Q)}] \in [\mathbb{I}^{|U|} \rightarrow \mathbb{B}]$ is a tensor representation for the set of solution mappings $Q(g)$. Otherwise, the set is empty and thus represented by the empty tensor which has all values set to 0.

Basic Graph Pattern. Consider a BGP $B = \{B^{(1)}, ..., B^{(r)}\}$ and its set of used variables U. Let g and T be defined as above. A tensor representation of applying B to g, i.e., $B(g)$, is given by T' with $T'_{\langle l \in U \rangle} \leftarrow \bigtimes_{i \in \mathbb{I}_r} T[\mathbf{k}^{B^{(i)}}]_{\langle l \in B^{(i)} | l \in U \rangle}$.

[6] It may not be defined if t contains any resource that is not in $\mathrm{dom}\,(id)$.

Algorithm 1: Einstein notation over hypertries

```
   Input: A list of hypertries $O$, a list of
          subscripts to the hypertries $L$ and a
          subscript to the result $R$
   Output: A hypertrie or another tensor
           representation
1  einsum($O, L, R$)
2      $k \leftarrow \langle id(\epsilon), \dots, id(\epsilon)\rangle, |k| = |R|$
3      $r \leftarrow$ empty tensor of rank $|R|$
4      einsum_rek($O, L, R, k, r$)
5      return $r$
6  einsum_rek($O, L, R, k, r$)
7      $U \leftarrow \{\lambda \in \Lambda \mid \Lambda \in L\}$
8      if $U \neq \emptyset$ then
9          $l \leftarrow$ any label from $U$
10         $L' \leftarrow \langle \Lambda \setminus l \mid \Lambda \in L\rangle$
11         $\mathcal{P} \leftarrow \langle \{i \mid \Lambda[i] = l\} \mid \Lambda \in L\rangle$
12         $K \leftarrow \bigcap_{j\in\mathbb{I}_{|O|}} \bigcap_{i\in\mathcal{P}[i]} \mathrm{dom}\,(c_i^{O[j]})$
13         for $\kappa \in K$ do
14             $O' \leftarrow \langle\rangle$
15             for $i \in \mathbb{I}_{|O|}$ do
16                 $s \leftarrow s[i] := \begin{cases} \kappa, \text{ if } i \in \mathcal{P}[i] \\ :, \text{ otherwise} \end{cases}$
17                 $O' \leftarrow O' + \langle O[i][s]\rangle$
18                 if $z(O'[i]) = 0$ then
19                     continue with next $\kappa$
20             if $l \in R$ then
21                 $k[i] \leftarrow \kappa$ with $R[i] = l$
22             einsum_rek($O', L', R, k, r$)
23     else
24         $r[k] + \leftarrow \prod_{o\in O} o$
```

Projection. Let B, r, U, g, and T be as above; consider $U' \subseteq U$. The projection $\Pi_{U'}(B(g))$ is represented by the tensor $T''_{\langle l\in U'\rangle} \leftarrow \bigtimes_{i\in\mathbb{I}_r} T[\mathbf{k}^{B^{(i)}}]_{\langle l\in B^{(i)} | l\in U\rangle}$.

With this mapping, we can now implement the key operations of the SPARQL algebra using hypertries.

6.2 Tensor Operations on Hypertries

Hypertries support both slices and Einstein summation efficiently. The efficient evaluation of slices was described in Sect. 5. An algorithm to evaluate Einstein summation based on a worst-case optimal multi-join algorithm by [16] is given by Algorithm 1 and discussed in this section. The algorithm is structured in two functions, a recursion starter and a recursion. The *recursion starter* (ll. 1–5) takes a list O of hypertrie operands, a list L of subscripts for the operands and a subscript R for the result as input and returns the resulting tensor r. A subscript is represented by a sequence of labels. The recursion starter prepares the key k and the result tensor r, calculates the result by calling the recursion `einsum_rek` and returns the result.

The recursion (ll. 6–24) additionally takes k and r as parameters. It first selects a label l that is used in L (ll. 7+9). If there is such a label (l. 8), a new operand's subscript L' is calculated by removing l from L (l. 10). It is to be used in the next recursion level. Next, the intersection K (l. 12) of edge labels by all those dimensions of the hypertries O (l. 11) that are subscripted by l is calculated. Note that operand subscripts with repeating labels, e.g., $\langle ?x, ?x\rangle$ for a TP `?x :pred ?x`, are implicitly covered by the construction of $\mathcal{P}$ which stores for each operand all positions that are subscripted with l. For each $\kappa \in K$ (l. 13) the l-subscripted dimensions of operands in O are resolved by κ and the new operands stored to O' (ll. 14–17). If any of the new operands is empty, the current κ is skipped (ll. 18–19). Operands that are not subscripted by l are just copied. If R contains l, k is set to κ at the corresponding position (ll. 20–21).

A recursive call is issued with the modified operands O' and operands' subscript L' (l. 22). If there is no label left in L, the break condition is reached (ll. 23–24). At this point the operands are scalars. The product of the operands is calculated and added to the entry at k in the result tensor r (l. 24).

6.3 Processing Order

The Einstein summation encapsulates all joins into a single operation. Thus, join ordering is not required. Nonetheless, the order in which labels are selected in line 9 of Algorithm 1 is crucial for the actual processing time. Clearly, the worst-case search space for the result is a subset to the Cartesian product of all operands' non-zero entries' keys. Evaluating a label that occurs more than once at operands reduces the search space if the size of the cut K in line 12 of Algorithm 1 is smaller than its inputs. Assuming equal distribution in the subhypertries, an upper bound to the reduction factor by a label is given by the ratio of the size of K to the maximum number of children of dimensions subscripted with the label. Given a sequence of operands O and their sequences of labels L, we define the reduction factor for a label l, an operand $o \in O$ and its labels $\Lambda \in L$ by

$$\psi_{o,\Lambda}(l) = \begin{cases} \frac{\mathrm{m}^-_{O,L}(l)}{\mathrm{m}^+_{o,\Lambda}(l)} & \text{if } l \in \Lambda, \\ 1 & \text{otherwise.} \end{cases}$$

where, $\mathrm{m}^-_{O,L}(l) = \min\left(|\operatorname{dom}(c_j^{O[i]})| \;\middle|\; L[i][j] = l\right)$ is the minimal cardinality of dimensions of any operand subscripted with l and $\mathrm{m}^+_{o,\Lambda}(l) = \max\left(|\operatorname{dom}(c_j^o)| \;\middle|\; \Lambda[j] = l\right)$ is the maximum cardinality of dimensions of o subscripted with l. Thus, the full guaranteed reduction factor for l is given by $\Psi_{O,L}(l) = \prod_i \psi_{O[i],L[i]}(l)$. To reflect the observation that in practice K is mostly smaller than $\mathrm{m}^-_{O,L}(l)$, we additionally divide $\Psi_{O,L}(l)$ by the number of sets of different sizes used in the cut. We hereby assume two such sets to be equal if they have the same size. As $\Psi_{O,L}(l)$ can be computed efficiently, it is calculated in each recursive call for all label candidates l. The label with the smallest factor is chosen.

7 Evaluation

7.1 Experimental Setup

All experiments[7] were executed on a server machine with an AMD EPYC 7742, 1 TB RAM and two 3 TB NVMe SSDs in RAID 0 running Debian 10 and OpenJDK 11.0.6. Each experiment was executed using the benchmark execution framework IGUANA v3.0.0-alpha2 [5], which we chose because it is open-source and thus ensures that our experiments can be repeated easily.

Benchmarks. We chose WatDiv [3] to generate a synthetic benchmark, and FEASIBLE [23] – a benchmark generation framework which uses query logs – to generate

[7] The full setup is available as Ansible playbook at https://github.com/dice-group/tentris-paper-benchmarks/releases/tag/v1.0.

a benchmark on real-world data. We used datasets of varied sizes and structures (see Table 2) by choosing the 1 billion-triple dataset from WatDiv as well as the real datasets English DBpedia 2015-10[8] (681 M triples) and Semantic Web Dog Food SWDF (372K triples). We used all query templates for WatDiv.[9] The benchmark queries for DBpedia and SWDF were generated by using FEASIBLE on real-world query logs contained in LSQ [21]. FEASIBLE was configured to generate `SELECT` queries with BGPs and optional `DISTINCT`. Queries with more than 2^{20} results were excluded from all benchmarks to ensure a fair comparison.[10] Statistics on the queries[11] used are given in Table 3.

Table 2. 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.

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 3. Statistics on the queries used for each dataset. #Q stands for the number of queries used in our evaluation. The average and the min-max range in brackets are given for the number of triple patterns (#TP), the number of results (#R), and the average join-vertex degree (avg JVD). The absolute and relative frequencies (in brackets) are given for the number of distinct queries (#D) and for the number of queries with large results (>5000 results).

Dataset	#Q	#TP	#R	#D	avg JVD	>5000 results
SWDF	203	1.74 (1–9)	5.5 k (1–304 k)	124 (61%)	0.75 (0–4)	18 (8.9%)
DBpedia	557	1.84 (1–14)	13.2 k (0–843 k)	222 (40%)	1.19 (0–4)	73 (13.1%)
WatDiv	45	6.51 (2–10)	3.7 k (0–34 k)	2 (4%)	2.61 (2–9)	9 (20.0%)

Triple Stores. We chose triple stores that were openly available and supported at least `SELECT` queries with or without `DISTINCT` and BGPs. All triple stores were required to be able to load the three benchmarking datasets. Triple stores which were not able to load all experimental datasets had to be excluded from our experiments. The following triple stores were used in our evaluation: a) TENTRIS 1.0.4, b) Virtuoso Open-Source Edition 7.2.5.1, c) Fuseki (Jena TDB) 3.14.0, d) Blazegraph v2.1.4, e) GraphDB Free v9.1.1, f) TripleBit [28],[12] which uses matrices to store triples similar to TENTRIS's

[8] We used this version because of query logs being available for FEASIBLE.

[9] For each template one query was generated. Additionally, queries not projecting all variables were included with and without `DISTINCT`.

[10] Virtuoso has a limit of 2^{20} results for queries answered via HTTP (see issue https://github.com/openlink/virtuoso-opensource/issues/700).

[11] All queries can be found in the supplementary material.

[12] We extended TripleBit to support entering SPARQL queries via command-line interface directly. This modification was necessary to use TripleBit with IGUANA. Code available at: https://github.com/dice-group/TripleBit/releases/tag/2020-03-03.

tensor, g) RDF-3X 0.3.8 [15], which uses many indices similar to the hypertrie used by TENTRIS, and e) gStore commit 3b4fe58-mod[13] which stores all data in-memory like TENTRIS. All triple stores were allocated the same amount of RAM.

Benchmark Execution. We used two evaluation setups to cater for the lack of HTTP endpoints in TripleBit and RDF-3X. In the first setup, we executed a HTTP-based benchmark. Here, five stress tests with 1, 4, 8, 16 and 32 users were executed using the HTTP SPARQL endpoints of the triple stores TENTRIS, Virtuoso, Fuseki, Blazegraph, and gStore. For GraphDB we executed only the stress tests with one user because it does not support more than two parallel users in the free version. The second setup covered triple stores with a command-line interface (CLI). This benchmark simulated a single user because CLI does not support multiple concurrent users. The second setup was executed against TENTRIS, RDF3X and TripleBit. Like in previous works [5,22], we set the runtime of all benchmarks to 60 min with a 3-min. timeout. The performance of each triple store was measured using Queries per Second (QpS) for each client. In addition, we assessed the overall performance of each triple store by using an average penalized QpS (avg pQpS) per client: If a triple store failed to answer a query before the timeout or returned an error, then said query was assigned a runtime of 3 min.

7.2 Evaluation of Join Implementation

The performance of TENTRIS depends partially on the approach used to process joins. In our first series of experiments, we hence compared our default join implementation (see Sect. 6.3) with two other possible join implementations: 2-way joins (T2j) and a random label selection strategy (Tr). We used the HTTP-based benchmarks with one user. The results of this series of experiments is shown in Fig. 7. Our join implementation based on multi-way joins and label ordering strategy contributes substantially to the performance of TENTRIS. Our default TENTRIS is the fastest w.r.t. avg pQpS and median QpS on all datasets. T2j and Tr time out on several queries through the benchmarks and answer several queries from each benchmark more than an order of magnitude slower than the default TENTRIS. Hence, we use the default implementation of TENTRIS throughout the rest of the experiments.

7.3 Comparison with Other Approaches

Figure 4 shows the results of our HTTP evaluation on SWDF, DBpedia and WatDiv. For each number of clients tested in the HTTP evaluation, two vertically aligned plots are given: the first shows a boxplot of QpS and the mean QpS for single queries as points, while the second reflects the avg pQpS. Please note the log-scale of the boxplots. For a better comparison between the number of clients tested, Fig. 5 shows a plot for each dataset with the avg pQpS depending on the number of clients. Analogous

[13] As IGUANA requires SPARQL Protocol conformance, we fixed the HTTP request handling of gStore, i.e., parsing requests, naming of parameters, and response content-type. With respect to benchmark execution, we set the timeout to 3 min, and the thread limit to 32 and raised the total memory limit to 800 GB. Code available at: https://github.com/dice-group/gStore-1/releases/tag/3b4fe58-mod.

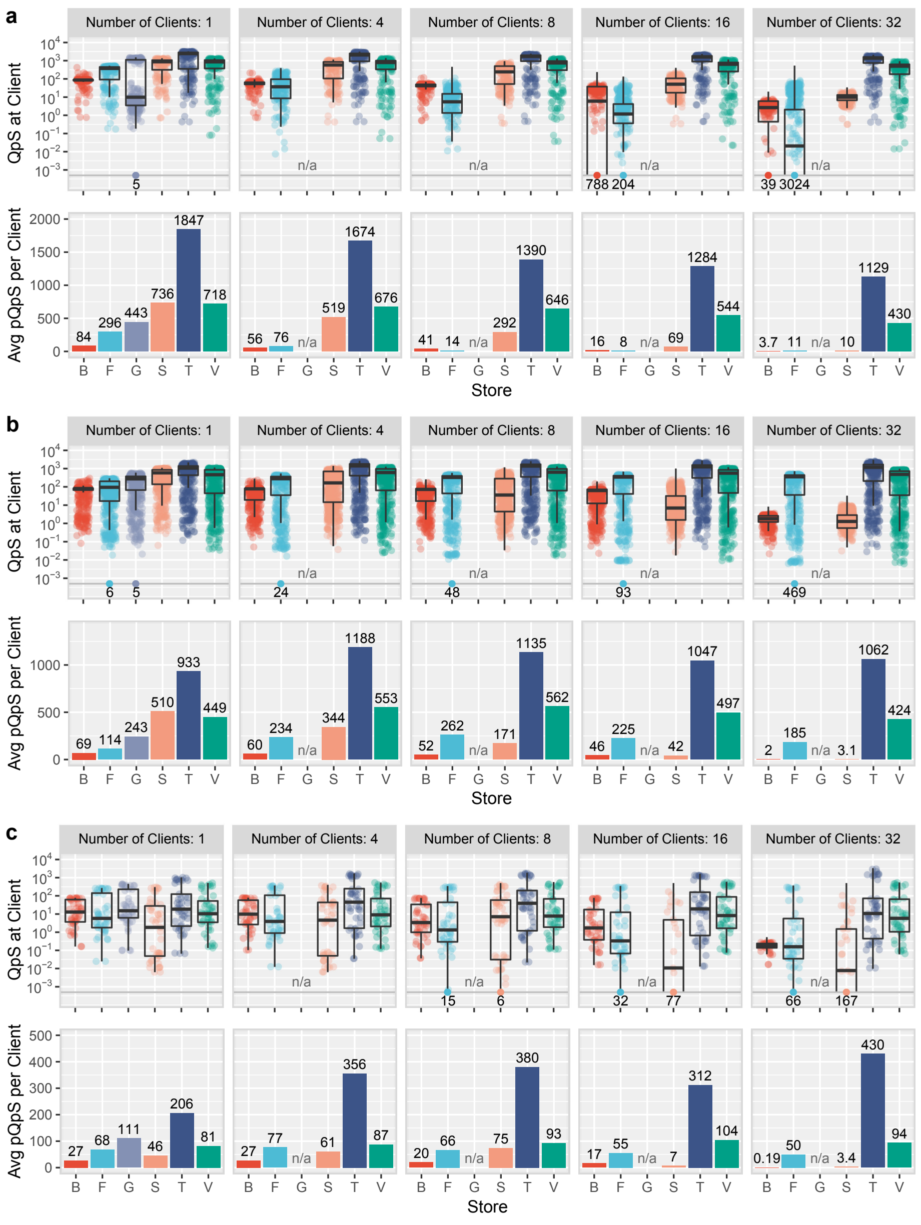

Fig. 4. Benchmark results on SWDF (a), DBpedia (b) and WatDiv (c) using HTTP with triple stores Blazegraph (B), Fuseki (F), GraphDB (G), gStore (S), TENTRIS (T) and Virtuoso (V): For each dataset, the first row shows boxplots for evaluations with 1, 4, 8, 16 and 32 clients respectively. Each point represents QpS for each single query, or mean QpS for a single query type for more than one client. For better readability we log-scaled the first line of the graphics. If queries with 0 QpS were present, those values were converted to $5 \cdot 10^{-4}$ QpS and the number of occurrences are shown as values on the bottom line. The second row shows avg pQpS per client.

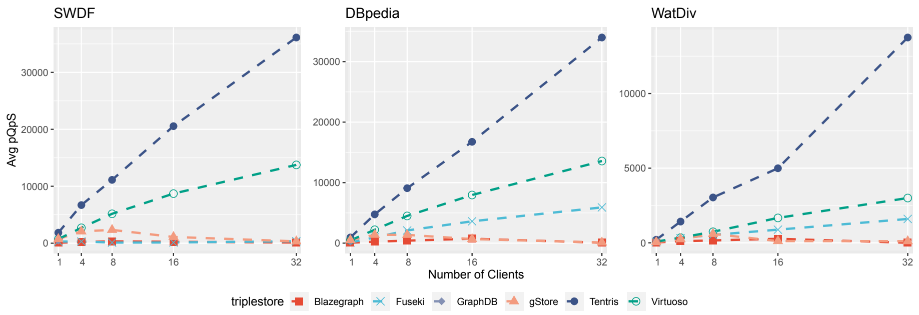

Fig. 5. The plots show for SWDF, DBpedia and WatDiv the avg pQpS for each triple store with increasing number of clients.

to Fig. 4, Fig. 6 provides the results of the CLI evaluation. Figure 7 shows the results of the comparison of different TENTRIS versions. For a comparison of the time and space requirements for loading the datasets into the triple stores see the supplementary material.

HTTP Results. Overall, TENTRIS outperforms all other triple stores for all datasets clearly with respect to avg pQpS. For a single client, our approach achieves a 1.83 to 2.51 times higher avg pQpS than the second best triple store, i.e., gStore or GraphDB. The avg pQpS of our approach is even 7.62 to 21.98 times higher than that of the slowest framework. With multiple users, TENTRIS scales almost linearly with the number of clients (see Fig. 5). TENTRIS is the only triple store in our evaluation that completed each query of all benchmarks at least once. Virtuoso succeeded on nearly all queries, with only a single failed query in the DBpedia benchmark with 32 users. The other triple stores failed on several queries across benchmark configurations.

As shown in Fig. 4a, TENTRIS is the fastest triple store for SWDF. It achieves avg pQpS that are at least 2 times higher than the second best and the median of its QpS lies above all values of all other stores. TENTRIS scales up the best to 32 users. The QpS per client drops from 1 to 32 clients by just 39%. Only Virtuoso shows a similar behavior, with a drop of 41%. The other triple stores are orders of magnitude slower when queried with multiple clients. Looking further into detail, TENTRIS outperforms the other stores for small queries which produce less then 5000 results (see Table 3) and Virtuoso is second best. For the 9% large queries with more than 5000 results, Blazegraph and gStore are about 1.5 times faster than TENTRIS for 1 client, but do not not scale with the number of clients; such that TENTRIS is 10 times faster than Blazegraph and 3 times faster than gStore for 32 clients.

For the DBpedia dataset, TENTRIS is the fastest store w.r.t. the avg pQpS and the median QpS. The result plots in Fig. 4b show that TENTRIS is almost two times faster than the second best triple store with respect to avg pQpS. It scales at least linearly with an increasing number of clients. When dividing the queries by small and large results, TENTRIS is always the fastest for DBpedia on small queries and only outperformed by gStore on large queries with 1–8 clients. Again, TENTRIS scales better and is fastest for 16–32 clients.

Like for the real-word datasets SWDF and DBpedia, TENTRIS outperforms the other triplestores on the synthetic WatDiv dataset by at least 1.8 times w.r.t. the avg pQpS. It scales at least linearly with an increasing number of clients. TENTRIS is fastest on the WatDiv dataset for small. For small queries and 1 client, GraphDB is the second best, while Virtuoso is the second best for multiple clients. For large queries, gStore is 1.5 times faster for 1 client than the second fastest TENTRIS, for 4 and 8 clients TENTRIS and gStore answer queries with roughly the same speed. With 16 to 32 clients, TENTRIS is the fastest at answering large queries by at least a factor of 2.

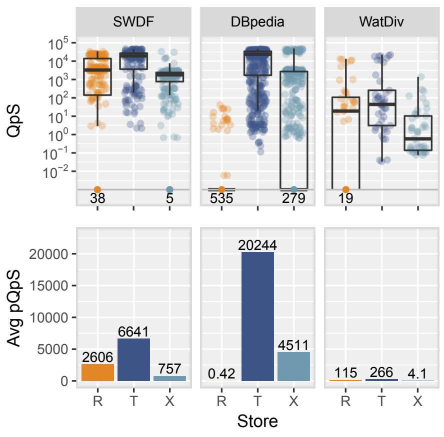

Fig. 6. Benchmarks on SWDF, DBpedia and WatDiv using a CLI with triple stores TripleBit (R), TENTRIS (T) and RDF-3X (X). A description of the layout is given in Fig. 4.

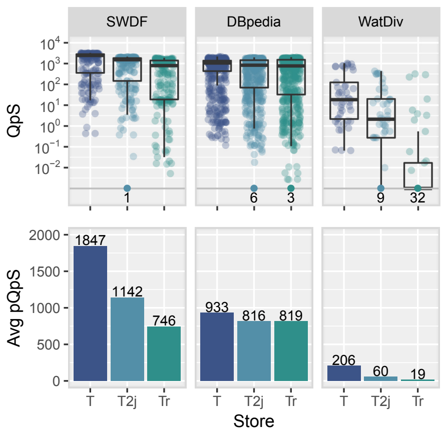

Fig. 7. Benchmarks on SWDF, DBpedia and WatDiv using different configurations of TENTRIS, i.e., the default configuration (T), using two-way joins (T2j) and using a random label order for the Einstein summation (Tr). A description of the layout is given at Fig. 4

CLI Results. The results of the CLI evaluation plotted in Fig. 6 show that TENTRIS clearly outperforms TripleBit and RDF-3X on all datasets. TripleBit and RDF-3X fail on 38 resp. 5 out of 203 queries for SWDF and 535 resp. 279 out of 557 queries for DBpedia. For the SWDF dataset, TENTRIS is at least 2.5 times faster with respect to pQpS. For DBpedia, the margin is even higher with 4.4-48,200 times higher pQpS. The scatterplot shows that TENTRIS answers more than 75% of the queries faster than TripleBit answers any query and than RDF-3X answers most queries. For the WatDiv dataset TENTRIS outperforms TripleBit and RDF-3X by at least 2.3 times w.r.t. pQpS.

Summary. Overall, TENTRIS outperforms all other approaches in the HTTP benchmarks client w.r.t. the average QpS per client across all datasets. The CLI experiments lead to a similar picture. While TENTRIS is always best for small queries with up to 5000 results, gStore is faster for large queries with more than 5000 results with up to 8 clients. This difference in performance seems to be due to the current result serialization of Tentris and will be addressed in future versions of the framework. The additional better scalability of the approach w.r.t. the number of clients suggests that Tentris is a viable alternative to existing solutions for querying RDF knowledge graphs. An analysis of our results suggests that the selection of the sequence of operations in the Einstein summation can be improved further by using heuristics (e.g., star joins vs. path joins) or by using function approximators ranging from regression-based solutions to deep learning.

8 Conclusion and Outlook

With Tentris, we present a time-efficient triple store for RDF knowledge graphs. We define a new mapping of RDF and SPARQL to tensors and tensor operations like slicing and Einstein summation. Our experimental results show that Tentris outperforms established triple stores with respect to QpS within experimental settings with up to 32 concurrent users. This improvement is the result of a novel tensor data structure, called hypertrie, that is designed to store low-rank tensors efficiently and allows the efficient evaluation of slices and Einstein summation. We show that hypertries allow for constant time slices on any combination of dimensions. An efficient evaluation of Einstein summation expressions on hypertries is achieved by an adaption of a worst-case optimal multi-join algorithm. Tentris will be extended in future works to be a fully-fledged triple store. Further improvements will include the data-driven improvement of the processing order for Einstein summation labels. Moreover, we will develop domain-specific versions of Tentris, e.g., geo-spatial extensions.

Acknowledgments. The authors would like to thank Lennart Austenfeld for automating the evaluation setup. This work has been supported by the German Federal Ministry for Economic Affairs and Energy (BMWi) within the project RAKI under the grant no 01MD19012D, by the German Federal Ministry of Education and Research (BMBF) within the project DAIKIRI under the grant no 01IS19085B, and by the EU H2020 Marie Skłodowska-Curie project KnowGraphs under the grant agreement no 860801.

References

1. Abdelaziz, I., Harbi, R., Khayyat, Z., Kalnis, P.: A survey and experimental comparison of distributed SPARQL engines for very large RDF data. Proc. VLDB Endow. **10**(13), 2049–2060 (2017)
2. Abraham, R., Marsden, J.E., Ratiu, T.: Manifolds, Tensor Analysis, and Applications. Springer, New York (1988)
3. 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
4. Atre, M., Chaoji, V., Zaki, M.J., Hendler, J.A.: Matrix "bit" loaded: a scalable lightweight join query processor for RDF data. In: WWW, pp. 41–50 (2010)

5. 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
6. De La Briandais, R.: File searching using variable length keys. In: Western Joint Computer Conference, IRE-AIEE-ACM 1959 (Western), pp. 295–298 (1959)
7. Einstein, A.: Die Grundlage der allgemeinen Relativitätstheorie. Annalen der Physik **354**, 769–822 (1916)
8. Erling, O.: Virtuoso, a hybrid RDBMS/graph column store. http://vos.openlinksw.com/owiki/wiki/VOS/VOSArticleVirtuosoAHybridRDBMSGraphColumnStore. Accessed 17 Mar 2018
9. Ermilov, I., Lehmann, J., Martin, M., Auer, S.: LODStats: the data web census dataset. In: Groth, P., et al. (eds.) ISWC 2016. LNCS, vol. 9982, pp. 38–46. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46547-0_5
10. Apache Software Foundation: Apache Jena documentation - TDB - store parameters (2019). https://jena.apache.org/documentation/tdb/store-parameters. Accessed 25 Apr 2019
11. Google Ireland Limited: tf.einsum — TensorFlow core r2.0 — TensorFlow (2019). https://pytorch.org/docs/stable/torch.html#torch.einsum. Accessed 06 Aug 2019
12. Jamour, F., Abdelaziz, I., Chen, Y., Kalnis, P.: Matrix algebra framework for portable, scalable and efficient query engines for RDF graphs. In: Proceedings of the Fourteenth EuroSys Conference 2019, EuroSys 2019. ACM (2019)
13. Kjolstad, F., Kamil, S., Chou, S., Lugato, D., Amarasinghe, S.: The tensor algebra compiler. In: Proceedings of the ACM on Programming Languages, 1(OOPSLA), October 2017
14. Motik, B., Nenov, Y., Piro, R., Horrocks, I., Olteanu, D.: Parallel materialisation of datalog programs in centralised, main-memory RDF systems. In: Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, AAAI 2014, pp. 129–137. AAAI Press (2014)
15. Neumann, T., Weikum, G.: RDF-3X: a RISC-style engine for RDF. Proc. VLDB Endow. **1**(1), 647–659 (2008)
16. Ngo, H.Q., Ré, C., Rudra, A.: Skew strikes back: new developments in the theory of join algorithms. SIGMOD Rec. **42**, 5–16 (2013)
17. Nickel, M., Tresp, V., Kriegel, H.-P.: Factorizing YAGO: scalable machine learning for linked data. In: WWW, pp. 271–280 (2012)
18. Noy, N., Gao, Y., Jain, A., Narayanan, A., Patterson, A., Taylor, J.: Industry-scale knowledge graphs: lessons and challenges. Queue **17**(2), 48–75 (2019)
19. Inc. Ontotext USA: Storage – GraphDB free 8.9 documentation. http://graphdb.ontotext.com/documentation/free/storage.html#storage-literal-index. Accessed 16 Apr 2019
20. MMG Ricci and Tullio Levi-Civita: Méthodes de calcul différentiel absolu et leurs applications. Mathematische Annalen **54**(1–2), 125–201 (1900)
21. Saleem, M., Ali, M.I., Hogan, A., Mehmood, Q., Ngomo, A.-C.N.: LSQ: the linked SPARQL queries dataset. In: Arenas, M., et al. (eds.) ISWC 2015. LNCS, vol. 9367, pp. 261–269. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25010-6_15
22. Saleem, M., Kamdar, M.R., Iqbal, A., Sampath, S., Deus, H.F., Ngomo Ngomo, A.-C.: Big linked cancer data: Integrating linked TCGA and PubMed. JWS (2014)
23. 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
24. SYSTAP, LLC. Bigdata Database Architecture - Blazegraph (2013). https://blazegraph.com/docs/bigdata_architecture_whitepaper.pdf. Accessed 29 Nov 2019
25. The SciPy community: numpy.einsum – NumPy v1.17 manual (2019). https://docs.scipy.org/doc/numpy/reference/generated/numpy.einsum.html. Accessed 6 Aug 2019

26. Torch Contributors: torch – PyTorch master documentation (2018). https://pytorch.org/docs/stable/torch.html#torch.einsum. Accessed 06 Aug 2019
27. De Virgilio, R.: A linear algebra technique for (de)centralized processing of SPARQL queries. In: Conceptual Modeling, pp. 463–476, October 2012
28. Pingpeng Yuan, P., Liu, B.W., Jin, H., Zhang, W., Liu, L.: TripleBit: a fast and compact system for large scale RDF data. Proc. VLDB Endow. **6**(7), 517–528 (2013)
29. 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)