TrieDF: Efficient In-memory Indexing for Metadata-augmented RDF

Olivier Pelgrin, Luis Galárraga, Katja Hose

TrieDF: Efficient In-memory Indexing for Metadata-augmented RDF

Olivier Pelgrin[1], Luis Galárraga[2], and Katja Hose[1]

[1] Aalborg University, Denmark {olivier,khose}@cs.aau.dk
[2] Inria, France luis.galarraga@inria.fr

Abstract. Metadata, such as provenance, versioning, temporal annotations, etc., is vital for the maintenance of RDF data. Despite its importance in the RDF ecosystem, support for metadata-augmented RDF remains limited. Some solutions focus on particular annotation types but no approach so far implements arbitrary levels of metadata in an application-agnostic way. We take a step to tackle this limitation and propose an in-memory tuple store architecture that can handle RDF data augmented with any type of metadata. Our approach, called TrieDF, builds upon the notion of tries to store the indexes and the dictionary of a metadata-augmented RDF dataset. Our experimental evaluation on three use cases shows that TrieDF outperforms state-of-the-art in-memory solutions for RDF in terms of main memory usage and retrieval time, while remaining application-agnostic.

1 Introduction

During the last 20 years, the Web has seen a proliferation of large collections of RDF data, i.e., triples ⟨ `subject`, `predicate`, `object` ⟩ describing real-world concepts ranging from common-sense to specialized domains. The triples are structured in what we call an *RDF graph* or *knowledge graph* (KG). KGs find applications in multiple AI-related tasks, such as question answering, information retrieval, smart assistants, etc.

Building and maintaining a large-scale KG is a titanic effort. It does not only require sophisticated RDF stores and collaborative tools, but also procedures and protocols to extract, cleanse, and integrate data from potentially heterogeneous sources. This is true regardless of whether the KG is manually or (semi-)automatically populated. A central aspect in KG construction is *metadata management*. RDF metadata includes, but is not limited to, provenance, validity intervals, spatial annotations, confidence statements, and versioning. As existing KGs grow and new initiatives come to existence, the need to manage statements about triples, i.e., RDF tuples, becomes more and more crucial.

There exist multiple solutions to represent metadata about RDF triples. Popular solutions are named graphs and reification. That said, these approaches are not free of limitations. A large number of fine-grained RDF graphs can be a challenge for quad stores [5]. Reification, on the other hand, quintuples the number of statements in a dataset; not to mention the fact that it also complexifies the queries. For these reasons, RDF engines support at most one level of metadata in an out-of-the-box fashion. This means that current stores can model statements about triples, but not statements about

quads, i.e., 5-tuples. They can neither model a versioned collection of graphs nor RDF statements originating from different sources with multiple validity intervals. Support for higher levels of metadata – equivalent to n-ary relationships – remains limited to very specific scenarios such as archiving [17].

This work takes a step to tackle the aforementioned limitations and proposes TrieDF, an in-memory RDF tuple store. TrieDF stores tuples of arbitrary length in a *trie*, an in-memory prefix-based tree originally used for compact storage and efficient retrieval of strings. TrieDF models *everything* as a trie, namely all indexes and the dictionary. Our evaluation shows that such an architecture yields a significant speed-up in retrieval with little penalty in memory consumption w.r.t. existing triple/quad stores. We also illustrate the utility of TrieDF at handling 5-tuples in the context of provenance and version management.

2 Preliminaries

RDF Graphs. An *RDF graph* $G \subseteq (\mathcal{I} \cup \mathcal{B}) \times \mathcal{I} \times \mathcal{T}$ is a set of triples $\langle$ `s`, `p`, `o` $\rangle$, with subject s, predicate p, and object o that model binary assertions about *entities*, for example, $\langle$ `:Denmark`, `:locatedIn`, `:Europe` $\rangle$. The sets $\mathcal{I}$, $\mathcal{B}$, and $\mathcal{T}$ are countably infinite sets of IRIs, blank nodes, and RDF terms respectively, with $\mathcal{T} = \mathcal{I} \cup \mathcal{B} \cup \mathcal{L}$ and $\mathcal{L}$ the set of literals. IRIs are web-scoped identifiers for entities such as `http://dbpedia.org/resource/Denmark` (abbreviated *:Denmark* for default prefix `http://dbpedia.org/resource/`); blank nodes are anonymous file-scoped identifiers; literals are non-referenceable data such as strings, numbers, and dates.

Metadata-augmented RDF Graphs. Given an RDF graph G, a metadata-augmented graph $\Gamma : G \rightarrow \mathcal{T}^n$ is an injective function that annotates each triple of the graph with a k-tuple of RDF terms. We can also see Γ as a set of n-tuples $q =\langle$ `s`, `p`, `o`, `...` $\rangle$ with $n = k + 3$. Metadata-augmented RDF graphs can model n-ary relationships in contrast to standard RDF graphs that can only model binary relationships.

3 Related Work

The need to store and manage metadata for RDF triples has given rise to a large literature body that we survey in two stages. First, we survey different techniques to encode metadata-augmented triples and store them in triple stores. In a second stage we discuss existing solutions to manage additional components in triples.

3.1 Encoding Metadata-augmented Triples

Reification is the process of encoding an n-ary statement through a set of binary relationships. For instance, consider the versioned triple $\langle$ `:Aalborg`, `:cityIn`, `:Denmark`, `3` $\rangle$ – that states that the triple is present in revision 3 of the graph. Under the standard reification, the triple is assigned a surrogate IRI (or blank node) u that is linked to all the components of the quad, resulting in 4 new triples: $\langle$ `u`, `:subject`, `:Aalborg` $\rangle$, $\langle$ `u`, `:predicate`, `:cityIn` $\rangle$, $\langle$ `u`, `:object`, `:Denmark` $\rangle$, and $\langle$ `u`, `:version`, `3` $\rangle$. Since

reification incurs a significant overhead both for storage and querying, other approaches have proposed more compact encoding strategies. The authors of [15] propose singleton properties as unique keys for statements in a context. In our example, such a context could be the graph revision where a triple occurs, e.g., ⟨ `:Aalborg`, `cityIn#3`, `:Denmark` ⟩. A more flexible scheme, called companion properties [10], proposes singleton properties per subject, for example, ⟨ `:Aalborg`, `cityIn#3.`*si*, `:Denmark` ⟩, where *si* is a local identifier that can be used to model subject-level metadata.

Reification and single properties have been used to encode one level of metadata, e.g., versioning, probabilities, provenance, temporal validity, etc. However, porting these strategies to scenarios with arbitrary levels of statement-centered metadata, e.g., provenance plus versioning, requires a careful application-dependent combination of the different schemes. This need has motivated the development of RDF-star [9], a data model that treats triples as first-class citizens and allows for nested statements such as ⟨ ⟨ `:Aalborg, :cityIn, :Denmark` ⟩, `:version`, `3` ⟩. Support for RDF-star is gaining traction in current RDF engines. Some commercial solutions such as RDFox, GraphDB, and Stardog can parse TriG, an RDF-star serialization text format. That said, none of those solutions can so far handle arbitrary levels of nestedness.

3.2 Beyond RDF Triples

RDF Named Graphs. Even though the RDF graph data model can be used to store metadata for RDF statements "natively", it was rather conceived as an analogy to documents and database tables. A named RDF graph is associated to an IRI g, and stores a presumably large collection of triples within a well-defined context, e.g., a particular data source. Nevertheless, named graphs have been used to store more fine-grained metadata such as validity intervals, revision numbers, and changesets [5, 17]. This can represent a challenge for classical RDF graph stores, such as Jena, Virtuoso, RDF4J, etc., that are not optimized for a large number of small RDF graphs. Since RDF named graphs cannot model metadata for quads, a few approaches have proposed highly specific solutions in the context of RDF/S inference [16].

Property Graphs. In this data model, both nodes (entities) and edges (relationships) can be assigned attributes, such as labels, timestamps, probabilities, sources, etc. Despite this flexibility, property graphs cannot store arbitrary levels of metadata for triples out of the box. Like named graphs, they rather provide a generic solution to store metadata about triples. This agnosticism has propelled adaptations of the graph model and existing engines (e.g., Neo4J, GraphDB) to particular applications, such as version and history management [7] and workflow provenance [6, 8, 12]. As for named graphs, solutions are application-dependent and not trivially portable to arbitrary settings.

Relational Databases. Notable designs to store RDF in tables are the three-column table and the entity-relationship model (a relation per class, a column per predicate). Storing metadata about RDF in a relational setting does not require any extension to the original data model, and standard engines provide efficient support for the most common metadata types such as temporal metadata [13], version control [11], and validity intervals. That said, the relational model is not optimized for the schema-free nature of RDF, which in the first place motivated the design of triple stores.

4 TrieDF

We now elaborate on TrieDF, our in-memory architecture to manage metadata-augmented RDF. TrieDF stores RDF tuples of arbitrary length in different indexes. The indexes do not store actual RDF terms but rather references to those terms in a space efficient dictionary (Figure 1b).

TrieDF borrows inspiration from tries – prefix-based trees for string storage. Nodes in a trie store single characters and each string is associated to a path in the tree. Strings with the same prefix, e.g., ***we****b*, ***we****ave*, ***we****asel*, share common nodes. TrieDF treats tuples as "strings" of items. Those items are either references to RDF terms (for indexes) or IRI chunks (for the dictionary). We elaborate on these use cases in the following sections.

4.1 Trie-based Indexes

Consider a version annotated RDF graph with quads $\langle$ s, p, o, v $\rangle$ such that v models the versions where the triple $\langle$ s, p, o $\rangle$ is present. Figure 1b depicts an SPOV index for the quads $\langle$ 1, 2, 3, 1 $\rangle$, $\langle$ 1, 2, 3, 2 $\rangle$, and $\langle$ 1, 5, 6, 2 $\rangle$ – the triples are encoded using a dictionary. We can infer that the triple $\langle$ 1, 5, 6 $\rangle$ was added in the second revision of the graph. We now describe the implementation of TrieDF and the operations it supports.

Implementation. Logically, each element of a tuple is associated to a node in the tree. Physically, nodes store only references to their children. Those references are organized in a red-black tree keyed by the value of the child node. If $|T|$ is the number of nodes in a trie T, an index in TrieDF has a space complexity of $O(|T|c_{max})$, where c_{max} is the highest out-degree of a node in T. This happens because red-black trees exhibit $O(n)$ space complexity in the number of keys.

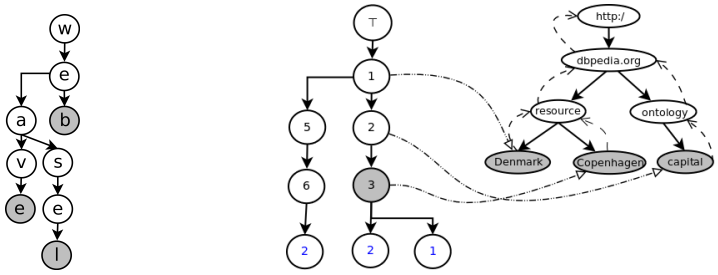
Tries are optimized for tuple lookup and prefix-based retrieval. These operations, as well as additions and deletions, are implemented as in standard tries. Therefore, looking up a tuple q incurs a time complexity of $O(|q|log(c_{max}))$.

Users of TrieDF can define indexes of arbitrary depth for all permutations of the elements of a tuple. Moreover, TrieDF can also operate as a standard triple store. In that case, the system stores triples in indexes SPO, POS, and OSP in line with standard engines. The nodes in the indexes store integers, precisely the memory addresses of the RDF terms in the dictionary [2], which is also implemented as a trie as explained next.

4.2 Trie-based Dictionary Encoding

Dictionary encoding maps the terms of an RDF dataset to an integer space for the sake of efficient space consumption and query processing. Dictionary encoding is often implemented using two hash tables, i.e., one for encoding (string to integer) and one for decoding (integer to string).

Even though dictionary encoding reduces memory consumption drastically for RDF engines, it does not tackle the inherent redundancy of RDF terms. For instance, common prefixes in IRIs are still stored multiple times. Similarly to the work of Bazoobandi et al. [2], the dictionary for IRIs is stored in an trie. In [2], a node is usually associated to a single character, although multiple nodes can be compressed (fusioned) into a single node if they form a single branch subtree. A drawback of such an approach is that



(a) A standard trie

(b) Trie-based index and dictionary

Fig. 1: Example tries

updates may cause fusioned nodes to split. In that case the tree must be rearranged, which increases update time. To make updates simpler – at the expense of some redundancy – TrieDF compresses IRIs by automatically coalescing all the nodes that lie between occurrences of the "/" character [21]. In other words, each node is logically associated to a chunk of an IRI as depicted in Figure 1b. If a node marks the end of an IRI, the memory address of the node serves as the integer identifier used by the tuple indexes described in the previous section.

In practice dictionary tries are bidirectional. That is, nodes store references to their children and parent. That way, a dictionary can retrieve the identifier associated to an IRI (lookup) and vice versa (reverse lookup). (Figure 1b). Because the benefit of a prefix-based representation is significantly less pronounced for literals [2, 20], they are currently stored in one-node tries.

5 Experiments

We evaluate TrieDF along three dimensions: loading time, main-memory consumption, and retrieval time (i.e., the time to return all the tuples that match a prefix). For this purpose, we test it in three use cases and compare it to other relevant in-memory solutions. Our scenarios cover a standard RDF graph, a versioned RDF graph (requiring quads), and a collection of 5-tuples.

TrieDF was written in C++. All experiments were run in a server with 256 GB of RAM, a 16-core CPU (AMD EPYC 7281), and an 8 TB HDD. The source code and experimental data is available at `https://relweb.cs.aau.dk/triedf`.

5.1 TrieDF for Triples

We first assess the performance of TrieDF when used as a standard in-memory triple store on DBpedia 2016-10 [1], YAGO 3.1 [18], YAGO 4 [19], and Wikidata [3]. For DBpedia we use the themes *mapping-based objects* and *mapping-based literals*. For YAGO 3.1 we consider the knowledge base's core, namely, the themes *facts*, *meta facts*, *literal facts*, *date facts*, and *labels*. For YAGO 4, we use the *facts* theme from the *English only* distribution. In regards to Wikidata we chose the *simple-statements* file of

the 2016-08 RDF export[3]. Details about the dataset sizes are available in the following table, in which we compare TrieDF with Jena[4] and RDFLib[5], two fully-fledged in-memory platforms for RDF/SPARQL management. They are widely used in production environments and offer mature and well-tested implementations.

	DBpedia	*YAGO 3.1*	*YAGO 4*	*Wikidata*
Triples	38M	85M	22M	138M
Size	4.9GB	12GB	3.0GB	17GB
RDF terms	11M	59M	8.5M	54M

Details about the experimental datasets for the triples evaluation.

Loading time. The following table shows the loading times of Jena, RDFLib, and TrieDF for the experimental datasets. Jena is consistently faster than the others systems, which can be explained by (i) a fast dictionary, (ii) a highly optimized RDF parser, and (iii) batching of insertions. Jena stores the dictionary in classical hash tables, which allows for constant time lookup and insertion of terms, in contrast to TrieDF that incurs a logarithmic lookup complexity. This, in contrast, optimizes for compactness (see paragraph on memory consumption). That said, TrieDF ranks second close to Jena, and outperforms RDFLib by a large margin.

	DBpedia	*YAGO 3.1*	*YAGO 4*	*Wikidata*
Jena	**587.14**	**1281.65**	**289.42**	**1665.48**
RDFLib	2816.16	7102.30	1626.85	9587.51
TrieDF	727.20	2105.37	358.20	2800.77

Loading time of the triples evaluation in seconds.

Retrieval time. We measure the average runtime of the different systems on single triple pattern queries of the following shapes: (i) ⟨ `s`, `p`, `?o` ⟩, (ii) ⟨ `?s`, `p`, `o` ⟩, (iii) ⟨ `s`, `?p`, `?o` ⟩, (iv) ⟨ `?s`, `?p`, `o` ⟩, and (v) ⟨ `?s`, `p`, `?o` ⟩ (? denotes a variable). For each query shape, we generated 200 queries by drawing the bound components randomly from the datasets, and then adding variables to the unbounded components as in [17]. The queries are implemented as classical retrieval operations, e.g., ⟨ `:Denmark`, `:capital`, `?o` ⟩ retrieves all tuples prefixed with ⟨ *:Denmark*, *:capital* ⟩ in the SPO trie index.

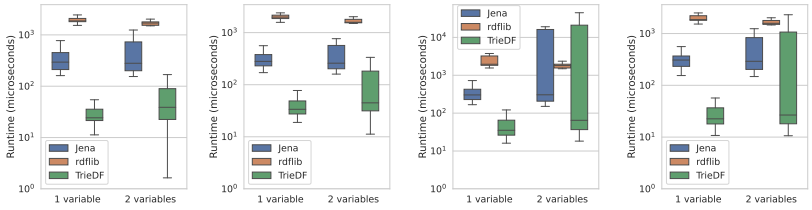
Figure 2 depicts the query runtime of the tested systems on the different datasets. The results show that TrieDF is at least one order of magnitude faster than the competitors for queries with one variable. When there are two variables, TrieDF still exhibits lower median runtimes, but its variance is large, specially for YAGO. This can be explained by the fact that those queries may sometimes have a large number of results. This phenomenon also affects Jena. While RDFLib is mostly insensitive to large result sets, it lags behind the other systems in terms of average retrieval time.

Memory consumption. Figure 3a shows the peek memory consumption of the different systems during loading and query execution. We observe that TrieDF outperforms Jena

[3] `http://tools.wmflabs.org/wikidata-exports/rdf/index.html`

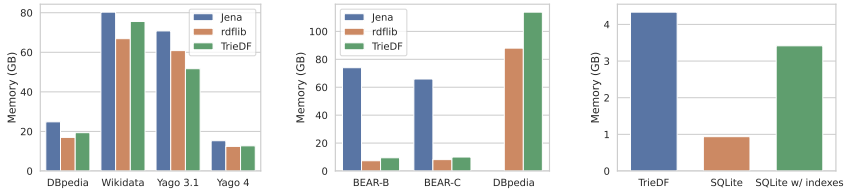
[4] `https://jena.apache.org/`

[5] `https://rdflib.readthedocs.org`



(a) DBpedia (b) Wikidata (c) YAGO 3.1 (d) YAGO 4

Fig. 2: Query runtime (microseconds) for triples queries (log scale)



(a) Triples (b) Quads (c) 5-tuples

Fig. 3: Peek memory usage in gigabytes (GB)

in all datasets, and all the competitors in YAGO 3. For the other datasets, our approach uses at most 13% more memory than RDFLib. This happens for two reasons. First, RDFLib's indexes are of fixed depth, which allows for some space savings: leaves do not need to accommodate for an additional pointer to its potential children. Second, RDFLib does not implement explicit dictionary encoding but rather relies on Python's internal variable handling (Python variables are actually keys pointing to their actual value in a hash table) to store references to RDF terms in the indexes. Relying on Python incurs important memory savings for RDFLib, however, the system remains two orders of magnitude slower than TrieDF at retrieval.

5.2 TrieDF for Quads

In this section we evaluate TrieDF at managing RDF quads $\langle$ s, p, o, v $\rangle$, that represent triples annotated with a revision number v. Our evaluation is based on the BEAR [4] benchmark datasets, and an archive consisting of the DBpedia versions from the 3.5 to the 2016-10 release, where one release is equivalent to one revision [17]. For each release, we use the same DBpedia themes as in our experiments with triples. As for BEAR, we use both the *BEAR-B* and *BEAR-C* datasets. We omitted *BEAR-A* because our competitors could not parse the input files due to formatting issues.

In order to provide versioning capabilities to TrieDF, we store RDF quads in SPOV, POSV, and OSPV indexes. We also compare TrieDF with the Jena in-memory models and RDFLib. For both competitors, we use named graphs to store each revision. This

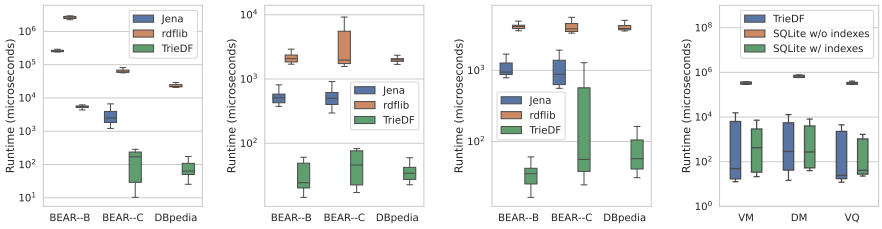
storage strategy, called *independent copies*, optimizes for data retrieval [4, 17] at the expense of high memory consumption.

Loading time. The table below shows the loading times of the evaluated systems. No results are provided for Jena in DBpedia, since the system runs out of memory. RDFLib lags behind Jena and TrieDF, with TrieDF being the fastest at loading the bulky revisions of BEAR-C, and Jena being the fastest at loading the more granular revisions of BEAR-B.

	DBpedia	*BEAR-B*	*BEAR-C*
Jena	-	**1094.83**	418.74
RDFLib	26433.76	4851.09	1663.40
TrieDF	**16074.17**	3743.55	**387.68**

Loading time of the quads evaluation in seconds.

Retrieval time. We measure the average runtime of the different systems on 100 single triple pattern queries of different types on versioned RDF graphs, namely version materialization (VM), delta materialization (DM), and version queries (VQ). The queries were randomly generated according to the experimental setup in [17].



(a) VQ queries (b) VM queries (c) DM queries (d) 5-tuples

Fig. 4: Query runtime for quads and 5-tuples (log scale)

Figure 4 shows the runtime of the evaluated systems for each type of query. We observe that TrieDF outperforms all the competitors by far, although the performance gap can vary drastically. In particular, TrieDF's indexes are optimized for VQ queries, for which they are 3 orders of magnitude faster than the competitors. Even though the independent copies approach used in Jena and RDFLib is optimal for VM queries (and to a lesser extent for DM queries), TrieDF is still one order of magnitude faster than the competitors.

Memory consumption. Figure 3b shows the peek memory consumption of the different systems during loading and query execution. We first highlight that Jena uses much more memory than the other systems. The reason is that Jena implements a classical independent copies approach, where each revision is entirety stored in a different graph. This leads to a lot of duplicated data. In contrast, RDFLib stores graphs (called contexts) in separate hash tables that map triples to graphs and graphs to triples. This mitigates redundancy. In the same vibe, TrieDF stores version identifiers in the last component of

each index, which reduces redundancy. While RDFLib showcases the lowest memory consumption, TrieDF strikes the best trade-off with at most 28% more peek memory usage than RDFLib in exchange for a speed-up of 3 orders of magnitude for retrieval.

5.3 TrieDF for 5-tuples

In this section we evaluate TrieDF on a 5-tuples setup where triples are annotated with provenance and version identifiers q, v, i.e., we store tuples $\langle$ s, p, o, q, v $\rangle$. We use a dump[6] of 27M tuples of the NELL [14] dataset. The NELL extractors collect metadata-augmented knowledge iteratively from the Web. This metadata includes, among other fields, confidence scores for the extracted triples, extraction sources, extraction methods, and the iteration of promotion, i.e., the iteration at which a triple is considered true and "officially" added to the knowledge base. We use the two latter fields in this evaluation. Since RDF storage systems do not support 5-tuples, we compare TrieDF against relational database systems with support for in-memory tables. After a comparison between SQLite and MariaDB, we chose the former due to its good performance in our setting. 5-tuples in TrieDF are represented via SPOQV, POSQV, and OSPQV indexes. We test SQLite with and without those indexes.

Loading time. We report loading times of 36.55s, 16.98s, and 47.27s for TrieDF, SQLite, and SQLite with indexes respectively. We observe that SQLite loads data significantly faster when no indexes are built, however when indexing is enabled, TrieDF is faster.

Retrieval time. We tested both systems on 100 queries of the same types defined for the quads evaluation with randomly bounded q and v. As suggested by Figure 4d, TrieDF achieves similar retrieval performance than an indexed 5-column SQLite in-memory table. The median runtime of TrieDF is better for VM and VQ queries.

Memory consumption. Figure 3c shows the peak memory usage of both SQLite and TrieDF when loading and querying the NELL dataset. We observe that indexing multiplies memory consumption by a factor of 3 in SQLite. TrieDF still uses 26% more memory than indexed SQLite, however TrieDF cannot leverage its trie-based dictionary to its full capacity. This happens because NELL does not use prefixed IRIs. Despite this rather suboptimal setting, TrieDF still shows comparable performance to SQLite.

6 Conclusion

We have presented an in-memory architecture based on tries to index and access annotated RDF triples efficiently. Our solution provides the user with a flexible architecture to manage arbitrary RDF metadata in main memory. Our experimental evaluation has shown that such as an approach strikes an interesting trade-off between retrieval time and memory footprint: it can yield a speed-up of up to 3 orders of magnitude in retrieval time in return to little (and sometimes no penalty) in memory usage. We believe that TrieDF is a first step towards a holistic solution to manage knowledge beyond RDF triples. As future work we envision to explore different strategies to reduce TrieDF's memory footprint. We also envision to couple our architecture with suitable in-disk storage and provide SPARQL query support.

[6] http://rtw.ml.cmu.edu/rtw/resources

Acknowledgements. This research was partially funded by the Danish Council for Independent Research (DFF) under grant agreement no. DFF-8048-00051B and the Poul Due Jensen Foundation.

References

1. Auer, S., Bizer, C., Kobilarov, G., Lehmann, J., Cyganiak, R., Ives, Z.G.: DBpedia: A Nucleus for a Web of Open Data. In: ISWC/ASWC. pp. 722–735 (2007)
2. Bazoobandi, H.R., de Rooij, S., Urbani, J., ten Teije, A., van Harmelen, F., Bal, H.E.: A Compact In-Memory Dictionary for RDF Data. In: ESWC (2015)
3. Erxleben, F., Günther, M., Krötzsch, M., Mendez, J., Vrandečić, D.: Introducing Wikidata to the Linked Data Web. In: ISWC. pp. 50–65 (2014)
4. Fernández, J.D., Umbrich, J., Polleres, A., Knuth, M.: Evaluating query and storage strategies for RDF archives. In: SEMANTICS. pp. 41–48 (2016)
5. Galárraga, L., Jakobsen, K.A., Hose, K., Pedersen, T.B.: Answering Provenance-Aware Queries on RDF Data Cubes Under Memory Budgets. In: ISWC (2018)
6. Galárraga, L., Mathiassen, K.A.M., Hose, K.: QBOAirbase: The European Air Quality Database as an RDF Cube. In: ISWC (Posters, Demos & Industry Tracks). CEUR Workshop Proceedings, vol. 1963. CEUR-WS.org (2017)
7. Haeusler, M., Trojer, T., Kessler, J., Farwick, M., Nowakowski, E., Breu, R.: ChronoGraph: A Versioned TinkerPop Graph Database. In: DATA (2017)
8. Hansen, E.R., Lissandrini, M., Ghose, A., Løkke, S., Thomsen, C., Hose, K.: Transparent Integration and Sharing of Life Cycle Sustainability Data with Provenance. In: ISWC. pp. 378–394 (2020)
9. Hartig, O.: Foundations of RDF⋆ and SPARQL⋆: An Alternative Approach to Statement-Level Metadata in RDF. In: AMW (2017)
10. Hernández, D., Hogan, A., Riveros, C., Rojas, C., Zerega, E.: Querying Wikidata: Comparing SPARQL, Relational and Graph Databases. In: ISWC (2016)
11. Huang, S., Xu, L., Liu, J., Elmore, A.J., Parameswaran, A.G.: OrpheusDB: Bolt-on Versioning for Relational Databases. PVLDB **10**(10), 1130–1141 (2017)
12. Kashliev, A.: Storage and Querying of Large Provenance Graphs Using NoSQL DSE. In: BigDataSecurity/HPSC/IDS. pp. 260–262 (2020)
13. Kaufmann, M., Fischer, P.M., May, N., Kossmann, D.: Benchmarking Bitemporal Database Systems: Ready for the Future or Stuck in the Past? In: EDBT (2014)
14. Mitchell, T.M., et al.: Never-Ending Learning. In: AAAI. pp. 2302–2310 (2015)
15. Nguyen, V., Bodenreider, O., Sheth, A.P.: Don't like RDF reification?: making statements about statements using singleton property. In: WWW (2014)
16. Pediaditis, P., Flouris, G., Fundulaki, I., Christophides, V.: On Explicit Provenance Management in RDF/S Graphs. In: TAPP (2009)
17. Pelgrin, O., Galárraga, L., Hose, K.: Towards Fully-fledged Archiving for RDF Datasets. Semantic Web Journal (2021)
18. Suchanek, F.M., Kasneci, G., Weikum, G.: YAGO: A Large Ontology from Wikipedia and WordNet. J. Web Semant. **6**(3), 203–217 (2008)
19. Tanon, T.P., Weikum, G., Suchanek, F.M.: YAGO 4: A Reason-able Knowledge Base. In: ESWC. pp. 583–596 (2020)
20. Umbrich, J., Hose, K., Karnstedt, M., Harth, A., Polleres, A.: Comparing data summaries for processing live queries over Linked Data. World Wide Web **14**(5-6), 495–544 (2011)
21. Yuan, P., Liu, P., Wu, B., Jin, H., Zhang, W., Liu, L.: TripleBit: a Fast and Compact System for Large Scale RDF Data. PVLDB **6**(7), 517–528 (2013)