In-Memory Dictionary-Based Indexing of Quoted RDF Triples

Ruben Taelman, Ruben Verborgh

IDLab, Department of Electronics and Information Systems, Ghent University – imec
Ghent, Belgium
E-mail: ruben.taelman@ugent.be

Abstract
The upcoming RDF 1.2 recommendation is scheduled to introduce the concept of *quoted triples*, which allows statements to be made about other statements. Since quoted triples enable new forms of data access in SPARQL 1.2, in the form of *quoted triple patterns*, there is a need for new indexing strategies that can efficiently handle these data access patterns. As such, we explore and evaluate different in-memory indexing approaches for quoted triples. In this paper, we investigate four indexing approaches, and evaluate their performance over an artificial dataset with custom triple pattern queries. Our findings show that the so-called *indexed quoted triples dictionary* vastly outperforms other approaches in terms of query execution time at the cost of increased storage size and ingestion time. Our work shows that indexing quoted triples in a dictionary separate from non-quoted RDF terms achieves good performance, and can be implemented using well-known indexing techniques into existing systems. Therefore, we illustrate that the addition of quoted triples into the RDF stack can be achieved in a performant manner.

KEYWORDS
RDF, RDF-Star, Indexing

1. Introduction

RDF [1] and Labeled Property Graphs (LPGs) [2] have been around in recent years as two major but diverging approaches for modeling Knowledge Graphs [3]. One of the main reasons for divergence, is the fact that LPGs allow datasets to contain statements about other statements, while RDF does not. This concept enables attaching metadata to statements, such as certainties or temporal validity. For example, it allows one to express *"Alice says that Violets are Blue."*, where the statement about Violets being Blue is *quoted* inside a statement about Alice.

In an effort to align these incompatibilities between RDF and LPGs, the RDF* [4] approach was introduced, which proposes an extension of the RDF data model and SPARQL query language with support for the concept of *quoted triples*. This approach was picked up by a W3C community group, and standardized in the RDF-star and SPARQL-star community group report [5]. This work is now being carried forward by the W3C RDF-star working group for standardization into the RDF 1.2 and SPARQL 1.2 recommendations.

Given the wide range of practical real-world applications [6, 7] for quoted triples, many open-source and commercial RDF and SPARQL systems have already implemented parts of this approach [8]. Notable are systems such as BlazeGraph, GraphDB, and Stardog, which offer the storage of quoted

7th Workshop on Storing, Querying and Benchmarking Knowledge Graphs (QuWeDa) at ISWC 2023
ruben.taelman@ugent.be (R. Taelman); ruben.verborgh@ugent.be (R. Verborgh)

triples in their triplestore, and enable queryable access using SPARQL. Even though some approaches offer reports of their systems passing RDF-star specification tests, and provide high-level documentation explaining the concepts of quoted triples for end-users, none of them offer detailed descriptions of their indexing approach, or query performance evaluations over them. As such, there is an open knowledge gap on the research question *"How to index quoted triples, and what is the impact on ingestion, storage, and query performance?"*.

To fill this gap, we explore four different indexing approaches, implement them in a single system for fair comparison, and evaluate them in terms of ingestion time, storage size, and query performance. We focus on in-memory indexing, but approaches are generalizable to mixed disk/memory approaches such as HDT [9]. Like many RDF indexing approaches [10, 11], we focus on providing indexed access for triple pattern queries, as these form the basis for answering full SPARQL queries.

2. Related Work

Given the recent introduction of RDF-star and the concept of quoted triples, scientific literature on making use of it is limited. First, several use cases [6, 7] have been explored using quoted triples. Furthermore, a declarative language called *RML-star* [12] has been introduced that allows heterogeneous datasources such as CSV and JSON to be mapped into RDF datasets containing quoted triples. This language is an extension of the RML language [13], and has been implemented in Morph-KGCstar [14]. Similarly, RSP-QL* [15] was introduced as an extension to the RSP-QL model [16] for RDF Stream Processing to support quoted triples. Finally, two approaches [17] are identified to transform RDF datasets containing quoted triples into a property graphs model [2].

RDF-star is seeing wide adoption among SPARQL implementations, for which a full list of implementations that adhere to the RDF-star community group specification can be found in [8]. Unfortunately, none of these approaches clearly document their storage and indexing approach, which motivates the need for this article on comparing various indexing techniques.

3. Background

Indexing is an important and well-understood element of RDF storage systems and SPARQL query engines, where it provides a trade-off between query execution time, storage space, and ingestion time. Existing approaches are either based on existing database technologies, such as relational databases [18] or document stores [19], or provide native support for RDF triples. In the context of this paper, we focus on the latter. Furthermore, we limit our discussion to the storage of RDF triples without considering the concept of named graphs, as these can be considered as fourth element in a quad-like structure, for which straightforward index extensions are possible.

3.1. Indexes For Different Orders

A first important concept in RDF indexing is the *storage of triples in different orders* [20], which is done by many RDF storage techniques, such as RDF-3X [10] and Hexastore [11]. Given that a triple consists of a subject (`S`), predicate (`P`) and object (`O`), both systems include six indexes for different triple component orders (`SPO`, `SOP`, `OSP`, `OPS`, `PSO` and `POS`). The presence of these indexes allows all possible triple patterns to be executed efficiently. For example, the triple pattern query `??O` can be answered most efficiently using the `OSP` or `OPS` indexes, while the query `S?O` could be answered using `SOP` and `OSP`. Next to triple pattern access efficiency, these orders also enable more efficient triple pattern join processing inside query engines, where the highly efficient sort-merge join could for example be used for joins between triple patterns if triples are sorted in the same manner. RDF-3X goes a step further, and also provides six aggregated indexes (`SP`, `SO`, `PS`, `PO`, `OS`, and `OP`), and three one-valued indexes (`S`, `P`, and `O`). The triples inside each index can be stored in different ways, such as ordered lists (Hexastore) or B+Trees (RDF-3X). Approaches such as HDT [9] and OSTRICH [21] go the different direction, and store fewer indexes (`SPO`, `POS`, `OSP`) to focus purely on the triple pattern access efficiency in combination with a lower storage cost. In the context of this article, we assume tree-like indexing via nested hash maps, and we refer to the triple component parts of an index as *triple component indexes*, corresponding to the levels in the tree. For example, the `SPO` index would have 3 triple component indexes: `S`, `P`, and `O`.

3.2. Dictionary Encoding

A second important aspect in RDF indexing is the *encoding of RDF terms using dictionaries*. The main purposes of dictionary-encoding are the reduction in storage overhead if RDF terms are reused across multiple triples inside indexes, and more efficient comparison of RDF terms during query processing. The dictionary itself is a datastructure that maintains a bidirectional mapping of RDF terms to their encodings. Instead of storing RDF terms directly inside indexes, terms are first encoded into a more compact datatype, such as an integer, which is then stored inside the index instead. At query time, non-variable triple pattern terms can also be encoded, and queried inside the index. When returning query results, encoded triples can be decoded using the dictionary.

Fig. 1 shows an illustration of the typical components of a triplestore. This example store contains three indexes, with triples stored in `SPO`, `POS`, and `OSP` orders in tree-like structure. These indexes make use of a single shared dictionary, which encodes the RDF terms inside all RDF triples stored by the indexes.

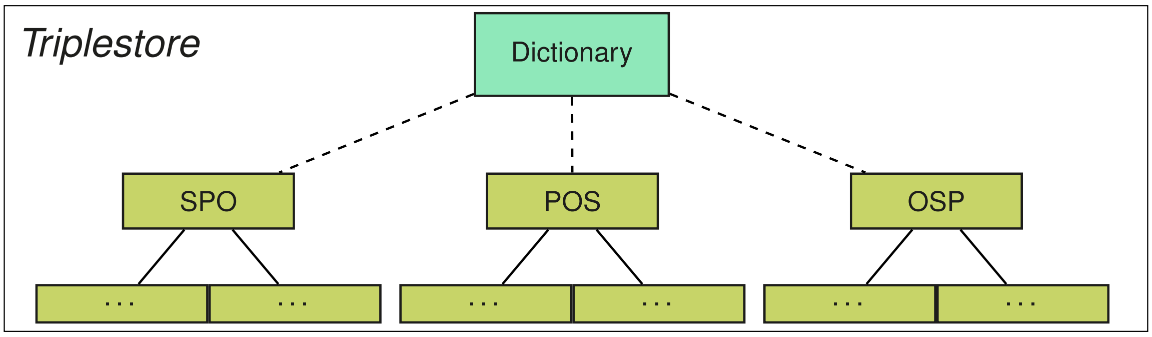


Fig. 1: The different components of a triplestore, containing one dictionary that is used by three indexes.

3.3. Triple Pattern Queries

To simplify discussions involving triple pattern queries, we outline a traditional high-level query execution approach for triple pattern queries in Algorithm 1, Algorithm 2, and Algorithm 3. As shown in Algorithm 1, the first step involves determining the most suitable index for a given triple pattern query. For example, a `??O` query can be answered most efficiently using the `OSP` index, because `O` can be selected in constant time from the root of the tree. The triple pattern query is then delegated to the index, which is executed according to Algorithm 2. In this step, we recursively drill down into the tree-like index by iterating over all matching terms of each triple pattern component. The algorithm for finding all matches for a single triple pattern component is shown in Algorithm 3, which either returns all terms in this part of the index if the term is a variable, or returns the term itself if the term is inside the index if the term is not a variable. We will replace parts of these algorithms when introducing our quoted triples indexing approaches in Section 5.

```
FUNCTION QueryStore(store, tp)
  INPUT:
    store: triple store
    tp: triple pattern
  OUTPUT:
    ts: sequence of triples
idx = most suitable index from store.indexes
ts = QueryIndex(idx, store.dict, tp)
RETURN ts
```

Algorithm 1: Pseudocode for executing a triple pattern query over a triplestore containing one or more indexes.

```
FUNCTION QueryIndex(idx, dict, tp)
  INPUT:
    idx: triple pattern index
    dict: dictionary
    tp: triple pattern
  OUTPUT:
    ts: sequence of triples
tpe = dict.encode(tp);
ts = [];
FOR subject IN QueryIndexComponent(idx, dict, tpe.subject])
  indx_s = idx[subject]
  FOR predicate IN QueryIndexComponent(indx_s, dict, tpe.predicate])
    indx_p = indx_s[predicate]
    FOR object IN QueryIndexComponent(indx_p, dict, tpe.object])
      object = indx_p[object]
      ts.push(subject, predicate, object)
    END
  END
END
RETURN ts
```

Algorithm 2: Pseudocode for executing a triple pattern query over an index, sorted in `SPO` order.

```
FUNCTION QueryIndexComponent(idxc, dict, term)
  INPUT:
    idxc: a certain triple component of a triple index
    dict: dictionary
    term: a term inside a triple pattern
  OUTPUT:
    ts: sequence of triples
matchingTerms = []
IF term.type === 'Variable'
  FOR key IN idxc
    matchingTerms.push(dict.decode(key))
  END
ELSE
  IF idxc contains dict.encode(term)
    matchingTerms.push(term)
  END
END
RETURN matchingTerms
```

Algorithm 3: Pseudocode for finding all matches of a single triple component inside an index.

4. Use Case

As example use case to illustrate different indexing approaches, we introduce a fictional dataset containing statements from different people on the color of violets in Listing 1.

```
@prefix : <http://example.org/foo#> .
:Alice :says << :Violets :haveColor :Blue   >> .
:Bob   :says << :Violets :haveColor :Yellow >> .
```

Listing 1: Turtle snippets containing statements by Alice and Bob on the color of violets.

To find out what color each person says violets have, we can execute the SPARQL query from Listing 2. This query contains a variable in the subject position, and a variable inside the quoted triple pattern of the object position. For brevity in the remainder of this article, we refer to the variables inside quoted triple patterns as *quoted variables*.

```
PREFIX : <http://example.org/foo#>
SELECT ?person ?color WHERE {
  ?person :says << :Violets :haveColor ?color >> .
}
```

Listing 2: SPARQL query to find what color each person says violets have.

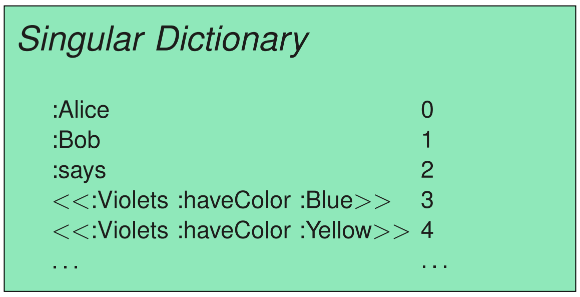
More advanced datasets may also contain *nested* quoted triples, in which any term inside a quoted triple could be another quoted triple. There is no upper limit of the nesting depth that can occur.

5. Indexing Approaches

In this section, we introduce four approaches for indexing quoted triples, with increasing levels of complexity. These approaches build upon the well-established methods of using dictionary encoding and storing triples in different orders as explained in Section 3. Our approaches only rely on changes within the dictionary mechanism, while the triple index itself can remain unchanged.

5.1. Singular Dictionary

A straightforward way to achieve dictionary encoding of quoted triples, is to include quoted triples directly inside the dictionary with all other RDF terms. As such, quoted triples are handled in exactly the same manner as other RDF term types. For dictionaries that map strings to integers, this requires a mechanism to convert quoted triples into strings. Fig. 2 shows an example of such dictionary contents based on our use case data.



:Alice	0
:Bob	1
:says	2
<<:Violets :haveColor :Blue>>	3
<<:Violets :haveColor :Yellow>>	4
...	...

Fig. 2: Plain terms and quoted triples are stored inside the same dictionary.

Executing triple pattern queries is identical to the baseline approach as explained in Section 3, except for triple patterns containing quoted variables, such as the query `?person :says <<Violets haveColor ?color>>`. As this approach has no direct way of matching the `?color` variable to quoted triples, we are required to convert quoted triple pattern terms containing variables to variables, and perform a post-processing step to only emit those triples that match the quoted triple pattern. The pseudo-code of this algorithm is shown in Algorithm 4.

```
FUNCTION QueryIndexSingularDict(idx, dict, tp)
  INPUT:
    idx: triple pattern index
    dict: singular dictionary
    tp: triple pattern
  OUTPUT:
    ts: sequence of triples
IF tp.subject.type === 'Quoted' && tp.subject contains variables
  s_filter = tp.subject
  tp.subject = new Variable()
IF tp.object.type === 'Quoted' && tp.object contains variables
  o_filter = tp.object
  tp.object = new Variable()
ts = QueryIndex(idx, dict, tp);
ts = ts.filter(t =>
  (s_filter === undefined || s_filter matches t.subject) &&
  (p_filter === undefined || p_filter matches t.predicate) &&
  (o_filter === undefined || o_filter matches t.object)
)
RETURN ts
```

Algorithm 4: Pseudocode of the algorithm for executing triple pattern queries using a singular dictionary. This algorithm is a variant of `QueryIndex` from Algorithm 2.

The main advantage of this approach lies in its simplicity of implementation. However, we hypothesize two main disadvantages:

1. **Storage overhead**: Quoted triples with shared terms lead to a storage overhead, such as the duplicate storage of `:Violets` and `:haveColor` in Fig. 2.
2. **Slow quoted triple pattern execution**: When executing triple pattern queries with quoted variables, there is no indexed access to matching quoted triples, which can lead to query performance issues.

5.2. Quoted Triples Dictionary

In an attempt to cope with the two disadvantages of the singular dictionary approach, we can dedicate the storage of quoted triples to a separate dictionary, as shown in Fig. 3.

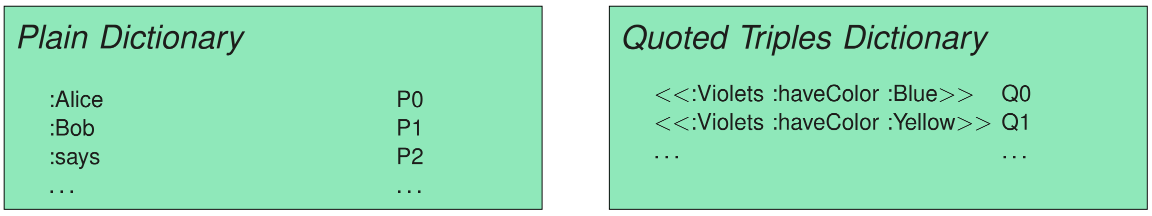

Plain Dictionary
:Alice P0
:Bob P1
:says P2
... ...
Quoted Triples Dictionary
<<:Violets :haveColor :Blue>> Q0
<<:Violets :haveColor :Yellow>> Q1
... ...

Fig. 3: Plain terms and quoted triples are stored in separate dictionaries.

To execute triple pattern queries in this approach, the post-processing step from the singular dictionary is not needed anymore. Instead, we can hook directly into the internal processing of separate triple component indexes. Concretely, when finding all matches of a given term inside a triple component index, we check if our term is a quoted triple pattern. If so, we perform an inner join between all quoted triple entries within the quoted triples dictionary, and the terms within the triple component index. If the triple component index is index in a hash-like manner, then this inner join can be done efficiently in a hash join manner. The pseudo-code of this algorithm is shown in Algorithm 5.

```
FUNCTION QueryIndexComponentQuotedDict(idxc, dict, term)
  INPUT:
    idxc: a certain triple component of a triple index
    dict: quoted triples dictionary
    term: a term inside a triple pattern
IF term.type === 'Quoted' && term contains variables
  matchingTerms = []
  FOR quotedTriple IN dict.getAllQuotedTriples()
    IF idxc contains dict.encode(quotedTriple)
      matchingTerms.push(quotedTriple)
    END
  RETURN matchingTerms
ELSE
  RETURN QueryIndexComponent(idxc, dict, term)
END
```

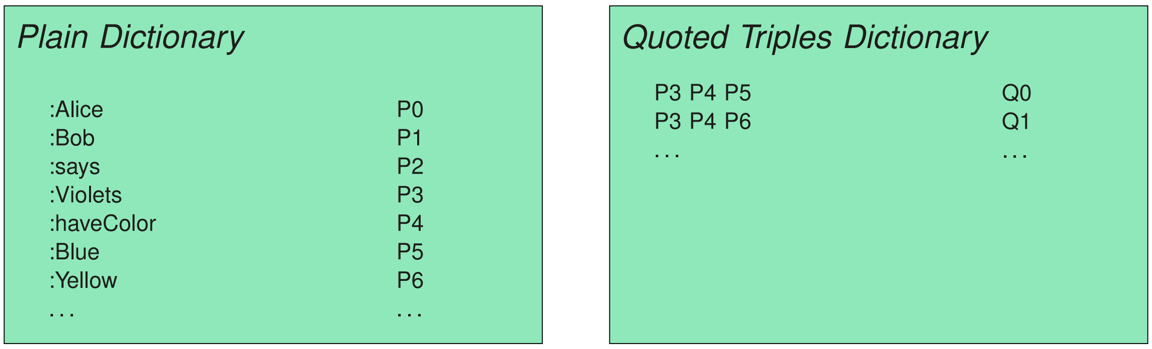
Algorithm 5: Pseudocode of the algorithm for finding all matching terms of a certain triple component inside an index using a quoted triples dictionary. This algorithm is a variant of `QueryIndexComponent` from Algorithm 3.

We hypothesize that this separated quoted triples dictionary will result in a lower storage overhead. Furthermore, we expect triple pattern execution to be faster due to the fact that a quoted triple pattern will only lead to matches with entries in the quoted triples dictionary, as opposed to *all* possible terms.

However, as shown in Fig. 3, this approach can still lead to redundant storage if terms are shared across different quoted triples, such as `:Violet` and `:haveColor`. Furthermore, the join inside the triple component index using all quoted triples might become too expensive if there are many non-matching quoted triples.

5.3. Referential Quoted Triples Dictionary

To solve the redundant storage issue within the quoted triples dictionary approach, we extend upon that approach by not storing quoted triples in full, but by instead encoding the three components of that quoted triples, and using those encodings as key inside the dictionary. Fig. 4 illustrates an example of this approach.



Plain Dictionary

:Alice	P0
:Bob	P1
:says	P2
:Violets	P3
:haveColor	P4
:Blue	P5
:Yellow	P6
. . .	. . .

Quoted Triples Dictionary

P3 P4 P5	Q0
P3 P4 P6	Q1
. . .	. . .

Fig. 4: Plain terms and quoted triples are stored in separate dictionaries, where quoted triples are recursively encoded using existing dictionary encodings.

The triple pattern query algorithm is identical to the one from Algorithm 5, except for the fact that dictionary encoding and decoding will require the extra step of encoding and decoding of the three triple components.

We hypothesize that this approach will lead to lower storage usage due to the shared encoding of redundant terms inside quoted triples.

5.4. Indexed Quoted Triples Dictionary

The Quoted Triples Dictionary approach requires triple component indexes to join with all quoted triples, which may be costly for selective quoted triple patterns in the presence of many non-matching quoted triples. To solve this problem, we can modify the storage of our Quoted Triples Dictionary from a map-like structure to a tree-like structure, so that triple pattern matching can be done more efficiently. Concretely, this tree-like structure can be implemented similar to a triple index, but it must map to integer encodings of quoted triples. Fig. 5 illustrates an example of this approach. This example only makes use of the `SPO` order for encodings, but in practise multiple other collation orders may be used.

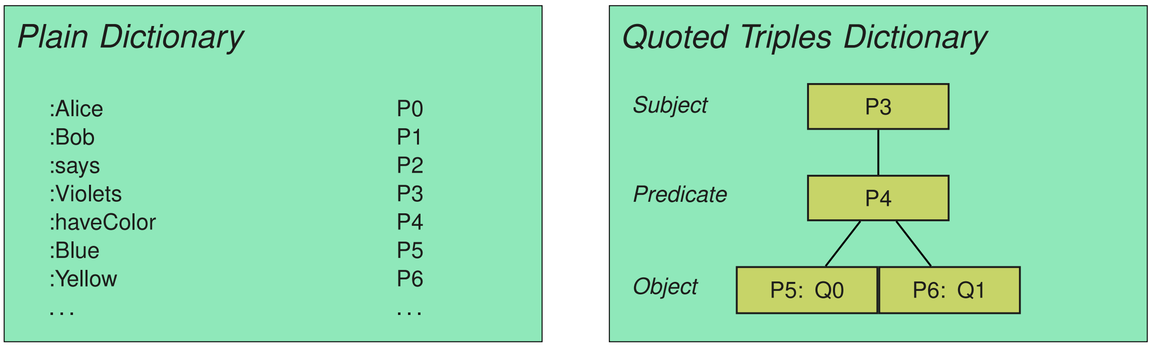


Fig. 5: Plain terms and quoted triples are stored in separate dictionaries, where quoted triples are recursively encoded using existing dictionary encodings, and indexed in a tree structure based on triple components.

Executing triple pattern queries is very similar to the approach of the Quoted Triples Dictionary, with the difference that instead of joining with *all* quoted triples, we join with only those quoted triples that match with the current quoted triple pattern. The pseudo-code of this algorithm is shown in Algorithm 6.

```
FUNCTION QueryIndexComponentQuotedDictIndex(idxc, dict, term)
  INPUT:
    idxc: a certain triple component of a triple index
    dict: quoted triples dictionary
    term: a term inside a triple pattern
IF term.type === 'Quoted' && term contains variables
  matchingTerms = []
  FOR quotedTriple IN dict.getMatchingQuotedTriples(term)
    IF idxc contains dict.encode(quotedTriple)
      matchingTerms.push(quotedTriple)
    END
  END
  RETURN matchingTerms
ELSE
  RETURN QueryIndexComponent(idxc, dict, term)
END
```

Algorithm 6: Pseudocode of the algorithm for finding all matching terms of a certain triple component inside an index using an indexed quoted triples dictionary. This algorithm is a variant of `QueryIndexComponent` from Algorithm 3.

We hypothesize that this approach will speed up triple pattern query performance due to the higher selectivity during joins between the quoted triples dictionary and the current component index.

6. Evaluation

In this section, we evaluate the impact of the indexing approaches discussed in Section 5 in terms of storage size, ingestion time, and query execution time. We start by discussing our implementation of the approaches, followed by our experimental setup, results, and end with a discussion.

6.1. Implementation

To achieve a fair comparison between the different indexing approaches, we have implemented all approaches in the same programming language (TypeScript/JavaScript). The implementation of these approaches is open-source, and is available on GitHub at https://github.com/rubensworks/rdf-stores.js.

6.2. Experimental Setup

To measure the performance impact of different quoted triple depths, we create synthetic datasets of various sizes. Our dataset generator is based on the data model of Section 4 with different people (size / 10) and colors (10), and allows any number of triples to be generated. Furthermore, it allows a *depth* parameter to be specified, which defines the nesting depth of quoted triples in object positions. For instance, a depth value of 1 generates quoted triples in the form of `?person :says << :Violets :haveColor ?color >>`, while a depth value of 3 generated quoted triples in the form of `?person :says << ?person :says << ?person :says << :Violets :haveColor ?color >> >> >>`.

For our experiments, we range the dataset from 1.000 to 1.000.000 triples, with the depth ranging from 1 to 5. For each combination, we measure the performance of the four indexing approaches in terms of the following metrics:

- **Storage size**: The total memory consumption after ingestion in MB.
- **Ingestion time**: The duration of ingesting the generated triples in milliseconds.
- **Query execution time**: The total duration of executing all triple pattern queries in milliseconds.

Query execution time was measured using 3 categories of queries (examples assume depth 2):

- **Low selectivity**: Query people in the form of: `?person :says << ?person :says << :Violets :haveColor :Red >> >>`. Each query produces size / 10 results.
- **Medium selectivity**: Query colors in the form of: `?person :says << :Bob :says << :Violets :haveColor ?color >> >>`. Each query produces 10 results.
- **High selectivity**: Query colors of specific people in the form of: `:Alice :says << :Bob :says << :Violets :haveColor ?color >> >>`. Each query produces 1 results.

The four indexing approaches were configured with three indexes (`SPO`, `POS`, `OSP`), and the indexed quoted triples dictionary was also configured with these three indexes. All experiments were executed on a MacBook Pro 13-inch, 2020 with 16GB of RAM and a 2,3 GHz Quad-Core Intel Core i7 processor. Our experimental setup is fully reproducible, and is available together with the raw results at https://github.com/rubensworks/experiments-indexing-quoted-triples.

6.3. Results

Fig. 6 and Fig. 7 respectively show the storage sizes and ingestion times for the different indexing approaches. Fig. 8, Fig. 9, and Fig. 10 respectively show the query execution times for low, medium, and high selectivity queries. We omit results for quoted triple depths that do not provide additional insights aside from the highest and lowest values. To show an overview of all storage sizes, all figures are logarithmic in both axes.

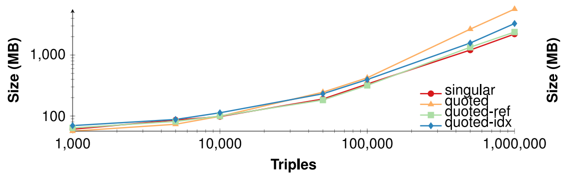
The labels inside each figure map to the indexing approaches as follows:

Singular Dictionary `singular`

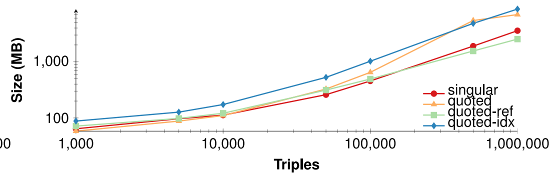
Quoted Triples Dictionary `quoted`

Referential Quoted Triples Dictionary `quoted-ref`

Indexed Quoted Triples Dictionary `quoted-idx`

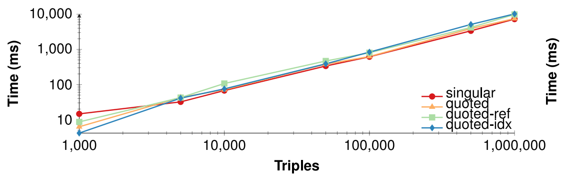


Subfig. 6.1: Depth 1

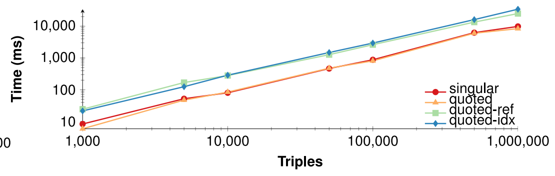


Subfig. 6.2: Depth 5

Fig. 6: Storage sizes for the 4 indexing approaches with increasing dataset sizes.

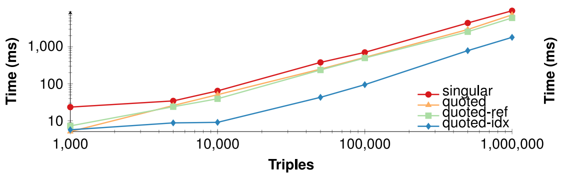


Subfig. 7.1: Depth 1

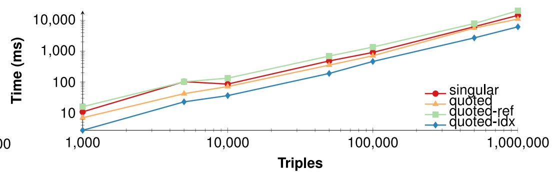


Subfig. 7.2: Depth 5

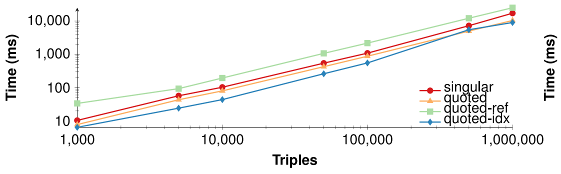
Fig. 7: Ingestion times for the 4 indexing approaches with increasing dataset sizes.



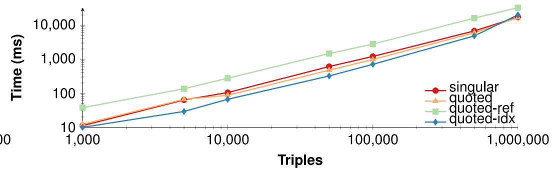
Subfig. 8.1: Depth 1



Subfig. 8.2: Depth 3

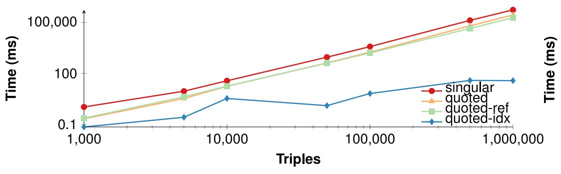


Subfig. 8.3: Depth 4

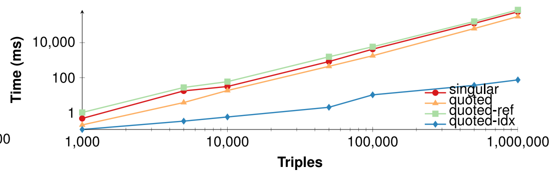


Subfig. 8.4: Depth 5

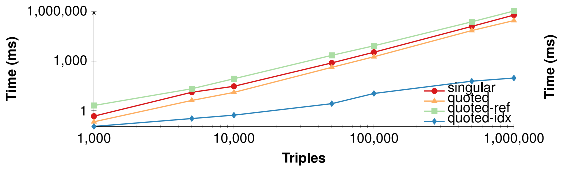
Fig. 8: Query execution times for the 4 indexing approaches with increasing dataset sizes with low result selectivity.



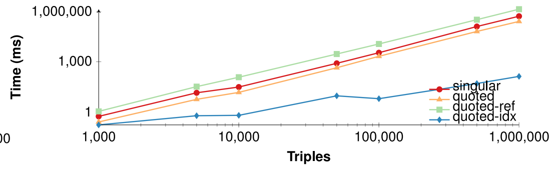
Subfig. 9.1: Depth 1



Subfig. 9.2: Depth 3

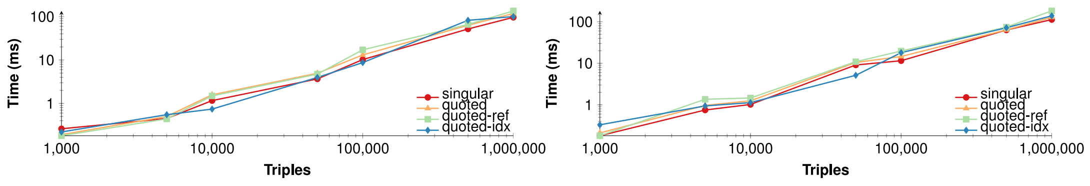


Subfig. 9.3: Depth 4



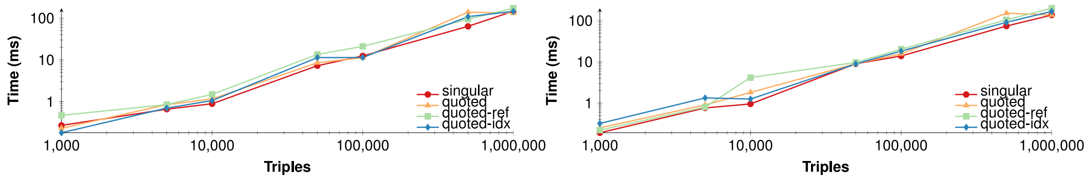
Subfig. 9.4: Depth 5

Fig. 9: Query execution times for the 4 indexing approaches with increasing dataset sizes with medium result selectivity.



Subfig. 10.1: Depth 1

Subfig. 10.2: Depth 3



Subfig. 10.3: Depth 4

Subfig. 10.4: Depth 5

Fig. 10: Query execution times for the 4 indexing approaches with increasing dataset sizes with high result selectivity.

6.4. Discussion

6.4.1. Storage Size

The results from Fig. 6 show that in terms of storage size, the singular dictionary and referential quoted triples dictionary approaches perform the best. The quoted triples dictionary and indexed quoted triples dictionary approaches on the other hand require significantly more memory. Contrary to what we hypothesized in Section 5, the storage overhead of the singular dictionary for terms inside quoted triples is less significant than expected, and the gains from the removal of storage redundancy with the referential quoted triples dictionary are minimal. This is due to the *string interning* memory optimization that is used by Node.js, whereby multiple equal string instances are only stored once in memory. The indexed quoted triples dictionary approach results in a significantly higher storage size due to the three indexes that are used to index quoted triples.

6.4.2. Ingestion Time

As expected, we observe similar results in terms of ingestion time in Fig. 7, where the singular dictionary and quoted triples dictionary are significantly faster than the referential and indexed quoted triples dictionary approaches. These approaches are faster due to their simpler encoding approach, whereas the referential and indexed quoted triples dictionary approaches require more operations during triple encoding.

6.4.3. Query Execution Time

The results in Fig. 8, Fig. 9, and Fig. 10 show that on average, the indexed quoted triples dictionary approach vastly outperforms all other approaches. This is most significant for triple patterns with medium selectivity due to the fact that this approach has indexes corresponding exactly to these queries, while the other approaches require iteration over all quoted triples. For triple patterns with low selectivity, the difference is smaller, but the indexed quoted triples dictionary approach is still faster overall. The difference for triple patterns with high selectivity is minimal, as the overhead of triple pattern dictionary encoding during query execution when fetching a single result becomes more apparent.

7. Conclusions

In this article, we discussed four approaches for the in-memory indexing of quoted triples, ranging from very naive to highly optimized for quoted triple pattern access.

Our results show that the indexed quoted triples dictionary approach is on average orders of magnitude faster than other indexing approaches. In the most extreme case, this approach is over 4.000 to 11.000 times faster than other indexing approaches for a dataset size of 1 million with quoted triples at depth 5. For smaller dataset sizes, lower quoted triple depths, and other types of queries, the difference becomes smaller. This significant speedup at query time comes with the expected cost of increased storage size and ingestion time, which is approximately two-fold for a large dataset.

As such, for use cases where quoted triple pattern queries occur at various depths, and an increase in storage size and ingestion time are acceptable, the indexed quoted triples dictionary approach is highly beneficial for achieving good query performance. Furthermore, given the fact that this approach stores quoted triple terms separate from all other terms inside the dictionary, there is no significant overhead of using such a dictionary by default in triplestores, even if is unknown before ingestion starts if quoted triples are present in the datasets.

In future work, there is a need to evaluate the performance of different indexing approaches over real-world data, as we only evaluated performance over artifically generated datasets, which do not capture real-world properties [22]. Furthermore, we wish to evaluate the performance of these indexes within full SPARQL queries by plugging them into a SPARQL query engine [23].

In conclusion, the outcomes of this work show that the inclusion of the concept of quoted triples into the RDF and SPARQL recommendations necessitates changes in triplestores in terms of indexing and dictionary encoding, but that approaches such as the indexed quoted triples dictionary are able to provide very good performance, while not requiring overly complicated additions implementation-wise.

References

[1] Cyganiak, R., Wood, D., Lanthaler, M.: RDF 1.1: Concepts and Abstract Syntax. W3C, http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/ (2014).

[2] Rodriguez, M.A., Neubauer, P.: Constructions from dots and lines. arXiv preprint arXiv:1006.2361. (2010).

[3] Hogan, A., Blomqvist, E., Cochez, M., d'Amato, C., Melo, G.de, Gutierrez, C., Kirrane, S., Gayo, J.E.L., Navigli, R., Neumaier, S., others: Knowledge graphs. ACM Computing Surveys (CSUR). 54, 1–37 (2021).

[4] Hartig, O.: Foundations of RDF* and SPARQL*:(An alternative approach to statement-level metadata in RDF). In: AMW 2017 11th Alberto Mendelzon International Workshop on Foundations of Data Management and the Web, Montevideo, Uruguay, June 7-9, 2017. Juan Reutter, Divesh Srivastava (2017).

[5] Hartig, O., Champin, P.-A., Kellogg, G., Seaborne, A.: RDF-star and SPARQL-star. W3C, https://www.w3.org/2021/12/rdf-star.html (2021).

[6] Kasenchak, B., Lehnert, A., Loh, G.: Use case: ontologies and RDF-star for knowledge management. In: The Semantic Web: ESWC 2021 Satellite Events: Virtual Event, June 6–10, 2021, Revised Selected Papers 18. pp. 254–260. Springer (2021).

[7] Braun, C.H.-J., Käfer, T.: Self-verifying Web Resource Representations Using Solid, RDF-Star and Signed URIs. In: The Semantic Web: ESWC 2022 Satellite Events: Hersonissos, Crete, Greece, May 29–June 2, 2022, Proceedings. pp. 138–142. Springer (2022).

[8] Group, R.D.F.-star W.C.C.: RDF-star Implementations. https://w3c.github.io/rdf-star/implementations.html (2023).

[9] Fernández, J.D., Martínez-Prieto, M.A., Gutiérrez, C., Polleres, A., Arias, M.: Binary RDF Representation for Publication and Exchange (HDT). Web Semantics: Science, Services and Agents on the World Wide Web. 19, 22–41 (2013).

[10] Neumann, T., Weikum, G.: RDF-3X: a RISC-style engine for RDF. Proceedings of the VLDB Endowment. 1, 647–659 (2008).

[11] Weiss, C., Karras, P., Bernstein, A.: Hexastore: sextuple indexing for semantic web data management. Proceedings of the VLDB Endowment. 1, 1008–1019 (2008).

[12] Delva, T., Arenas-Guerrero, J., Iglesias-Molina, A., Corcho, O., Chaves-Fraga, D., Dimou, A.: RML-star: A declarative mapping language for RDF-star generation. In: ISWC2021, the International Semantic Web Conference. CEUR (2021).

[13] Dimou, A., Vander Sande, M., Colpaert, P., Verborgh, R., Mannens, E., Van de Walle, R.: RML: A generic language for integrated RDF mappings of heterogeneous data. Ldow. 1184, (2014).

[14] Arenas-Guerrero, J., Iglesias-Molina, A., Chaves-Fraga, D., Garijo, D., Corcho, O., Dimou, A.: Morph-KGCstar: Declarative generation of RDF-star graphs from heterogeneous data. Semantic Web (Under Review). (2023).

[15] Keskisärkkä, R., Blomqvist, E., Lind, L., Hartig, O.: RSP-QL: Enabling Statement-Level Annotations in RDF Streams. In: Semantic Systems. The Power of AI and Knowledge Graphs: 15th International Conference, SEMANTiCS 2019, Karlsruhe, Germany, September 9–12, 2019, Proceedings. pp. 140–155. Springer (2019).

[16] Dell'Aglio, D., Della Valle, E., Calbimonte, J.-P., Corcho, O.: RSP-QL semantics: A unifying query model to explain heterogeneity of RDF stream processing systems. International Journal on Semantic Web and Information Systems (IJSWIS). 10, 17–44 (2014).

[17] Abuoda, G., Dell'Aglio, D., Keen, A., Hose, K.: Transforming RDF-star to Property Graphs: A Preliminary Analysis of Transformation Approaches–extended version. arXiv preprint arXiv:2210.05781. (2022).

[18] Erling, O., Mikhailov, I.: Virtuoso: RDF support in a native RDBMS. In: Semantic Web Information Management. pp. 501–519. Springer (2010).

[19] Wallgrün, J.O., Frommberger, L., Wolter, D., Dylla, F., Freksa, C.: Qualitative spatial representation and reasoning in the SparQ-toolbox. In: International Conference on Spatial Cognition. pp. 39–58. Springer (2006).

[20] Harth, A., Decker, S.: Optimized index structures for querying rdf from the web. In: Third Latin American Web Congress (LA-WEB'2005). pp. 10–pp. IEEE (2005).

[21] Taelman, R., Vander Sande, M., Van Herwegen, J., Mannens, E., Verborgh, R.: Triple Storage for Random-Access Versioned Querying of RDF Archives. Journal of Web Semantics. (2018).

[22] Duan, S., Kementsietsidis, A., Srinivas, K., Udrea, O.: Apples and oranges: a comparison of RDF benchmarks and real RDF datasets. In: Proceedings of the 2011 ACM SIGMOD International Conference on Management of data. pp. 145–156 (2011).

[23] Taelman, R., Van Herwegen, J., Vander Sande, M., Verborgh, R.: Comunica: a Modular SPARQL Query Engine for the Web. In: Proceedings of the 17th International Semantic Web Conference (2018).