Efficient Approach for Storage and Search of Large RDF Datasets Using Compressed Indexes

Pusala Rajeswari [1], P.T.Sirisha[2]
1 M.C.A Student, Department of Computer Applications, Sri Padmavathi college of Computer science and Technology
2 Assistant Professor, Department of Computer Applications, Sri Padmavathi college of Computer science and Technology

Abstract–The sheer increase in volume of RDF data demands efficient solutions for the *triple indexing problem*, that is to devise a compressed data structure to compactly represent RDF triples by guaranteeing, at the same time, fast pattern matching operations. This problem lies at the heart of delivering good practical performance for the resolution of complex SPARQL queries on large RDF datasets. In this work, we propose a trie-based index layout to solve the problem and introduce two novel techniques to reduce its space of representation for improved effectiveness.

Introduction:

Resource Description Framework (RDF[1]) is a W3C standard offering a general graph-based model for describing information as a set of (*subject*, *predicate*, *object*) relations, known as triples. Representing data in RDF allows subject and object entities to be unambiguously identified and connected through directed and explainable relationships, thus favoring the integration and reuse of different information sources. Although RDF was initially conceived as a metadata model for the Semantic Web and the Linked Data [1], its generality and flexibility favoured its diffusion in other domains ranging from digital libraries to bioinformatics and business intelligence. Moreover, the success of initiatives such as *schema.org* and *opengraph*[2] made RDF the de-facto standard format for publishing semi-structured information in social networks and Web sites. In fact, major search engines like Google and Bing are providing increasingly-better support for RDF.

Such wide popularity encouraged the development of several data management systems able to deal with large RDF datasets and the complexity of querying them via SPARQL[3], a query language that understands the RDF model and allows to select and join graph-structured data based on both content and patterns. Not surprisingly, the increasing volume of RDF data available on-line and in various repositories pushed researchers to investigate specific solutions enabling users and software agents to store, access and query RDF graph-structured data efficiently. In this direction we can identify four relevant research topics.

- *Compressed string dictionaries.* Each RDF statement has three components: a *subject* (S), an *object* (O), and a *predicate* (P, also called a *property*) that denotes a relationship. Each one of these components is a URI string (or even a literal in the case of an object). Since URI strings can be very long and the same URI generally appears in many RDF statements, the components of triples are commonly mapped to integer IDs to save space, so that each triple in the dataset can be represented with three integers.
- *Triple indexing data structures.* Indexes built over the set of triples should allow fast access to data for processing complex SPARQL queries involving large sequences of *triple selection patterns* over RDF graphs [2], [3].
- *Query-planning algorithms.* An effective query-planning algorithm has to find a suitable order to the set of atomic selection patterns that are needed to solve a SPARQL query, in order to speed up its execution and optimize expensive join operations [4], [5], [6].
- *Inference.* RDF triples are used to infer new relationshipsin order to improve the quality of the data and discoverpossible inconsistencies [7], [8].

In this work, we focus on the *triple indexing problem* that isto design a static index for the integer triples that attains to efficient resolution of all possible selection patterns using aslittle space as possible. This is crucial to guarantee practicalSPARQL query evaluation. Therefore, we do not directly manage a string dictionary mapping URIs to integer IDs that, as discussed above, is a different problem.

Moving from a critical analysis of the state-of-the-art, we note that existing solutions to the problem require toomuch space, because either: rely on materializing all possible permutations of the S-P-O components [9], [5]; use expensive additional supporting structures [10], [5]; do not use sophisticated data compression techniques to effectively reduce the space for encoding triple identifiers [11], [6]. Furthermore, this additional space overhead does not always.

pay off in terms of reduced query response time. The aimof this work is that of addressing these issues by proposing compressed indexes for RDF data that are both compact *and* fast.

contributions. In particular, detailed contributions are as follows.

1) propose the use of a *trie*-based index layout delivering a considerably better efficiency for all triple selection patterns, thanks to the cache-friendly nature of its pattern matching algorithm. Specifically, the index materializes three different permutations of the triples in order to (symmetrically) support all triple selection patterns with one or two wildcard symbols. By leveraging on well- engineered compression techniques, we show that this design is already as compact as

the most space-efficient competitor in the literature and 2 – 4 faster on average for all selection patterns.

2) Starting from the aforementioned index layout, we devise two optimization aimed at reducing the redundancy of the representation. The first technique builds on the observation that the order of the triples given by a permutation can be actually exploited to compress another permutation, hence *cross-compressing* the index. The second technique shows that it is possible to *eliminate* a permutation without affecting (or even improving) triple retrieval efficiency.
3) Extensive and thorough experiments aimed to assess the space and time performance of our proposal versus state-of-the-art competitors are conducted on publicly available RDF datasets with a number of triples ranging from 88 millions up to 2 billions and show that our best space/time trade-off configuration substantially outperforms existing solutions at the state-of-the-art, by taking 30 – 60% less space *and* speeding up query execution by a factor of 2 – 81×.

Source code. In the interest of reproducibility, our code is available at https://github.com/jermp/rdf_indexes.

The permuted trie index

In this section we introduce our trie-based index that solves the *triple indexing problem* mentioned in Section 1: compressing a large set of integer triples by granting the efficient resolution of sequences of triple selection patterns. In particular, in Section 3.1 we introduce the base indexing data structure and in Section 3.2 and 3.3 we discuss two variants aimed at reducing redundancies in the representation.

In order to better support our design choices and explain the intuition behind the described ideas, we show in the following the results of some motivating experiments conducted on the DBpedia dataset – "the nucleus for a Web of Data" [23] – that is the English version of DBpedia (version 3.9) containing more than 351 millions of triples. (See also Table 3 at page 9). In Section 4 we will report on the comprehensive set of experiments conducted to assess the performance in space and time of our implementations versus state-of-the-art competitors on publicly available RDF datasets of varying size and characteristics.

Data structure

As a high-level overview, our index maintains three different permutations of the triples, with each permutation sorted to allow efficient searches and effective compression. The permutations chosen are SPO, POS and OSP in order to (symmetrically) support all the six different triple selection patterns with one or two wildcard symbols: SP? and S?? over SPO; ?PO and ?P? over POS; S?O and ??O over OSP. The two additional patterns with, respectively,

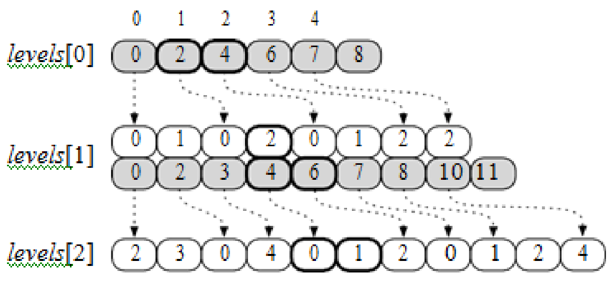

Fig. 1. A trie data structure representing a set of triples. Shaded boxes indicate pointers whereas the others refer to the nodes of the trie. Nodes in the first level are implicit, thus are not part of the data structure but reported here in smaller font for better visualization. Similarly, the dashed arrows are just for representational purposes and point to the position written in the corresponding originating box. Lastly, we highlight in thick stroke the nodes and pointers that are accessed during the resolution of the pattern (1, 2, ?).

all symbols specified or none, can be resolved over any permutation, e.g., over the canonical SPO in order to avoidpermuting back each returned triple.

Each permutation of the triples is represented as a 3-level trie data structure, with nodes at the same level con-catenated together to form an integer sequence. We keeptrack of where groups of siblings begin and end in theconcatenated sequence of nodes by storing such pointers asabsolute positions in the sequence of nodes. Therefore, thepointers are integer sequences as well. Moreover, since thetriples are represented by the trie data structure in sortedorder, the n node IDs in the first level of each trie are alwayscomplete sequences of integers ranging from 0 to n 1 and,thus, can be omitted. We can model each trie data structurewith an array *levels*[0, 1, 2] of three objects, each one havingtwo integer sequences of *nodes* and *pointers*. An exception isrepresented by *levels*[0] for which, as discussed above, nodesare missing, and by *levels*[2] for which pointers are missing.Refer to Fig. 1 for a pictorial example in which the fol- lowing set of triples is indexed: (0, 0, 2), (0, 0, 3), (0, 1, 0), (1, 0, 4), (1, 2, 0), (1, 2, 1), (2, 0, 2), (2, 1, 0), (3, 2, 1), (3, 2, 2), (4, 2, 4)}.

The advantage of the introduced layout is *two-fold*. First, we can effectively compress the integer sequences that constitute the levels of the tries to achieve small storage requirements. Second, as exemplified above, the triple selection patterns are made *cache-friendly* and, hence, efficient by requiring to simply *scan* ranges of consecutive nodes inthe trie levels. In what follows, we explore and quantify the impact of these two advantages.

Before continuing, an important consideration is in order. The described data structure is *static*, i.e., it does *not* directly support dynamic updates. Note, however, that a simple amortized solution could solve this limitation. For example, we could maintain a "small" index holding the most recent updates. Whenever the small index reaches a predefined size, its content is merged with the one of the main, static, index. Queries also need to involve both indexes and their results have to be merged accordingly.

Solving triple selection patterns. The pseudo code reported in Fig. 2 illustrates how triple patterns with one or two wildcard symbols are supported by our index. Given a

sequence S, function S.find(i, j, x) finds the ID x in the range $S[i, j)$ and returns its absolute position in the sequence. If x is not found in the range, a default position, e.g., 1, is returned to signal the event and the number of matches will be 0. Function S.iterator at(i) instead instantiates an iterator starting at $S[i]$. We assume that invalid iterator isa function returning an iterator over an *empty* range (that is invalid). Furthermore, the select algorithm creates two iterators to scan ranges of the second and third levels of the trie, respectively. These iterators are then used to define a final iterator that combines the iterating capabilities of both objects (line 20 of the pseudo code).

For example, pattern matching (1, 2, ?) will return the two triples (1, 2, 0) and (1, 2, 1) because these are the ones sharing the first two specified components (1, 2). Fig. 1 highlights the nodes and pointers accessed during the resolution of such pattern. In this case, we begin by fetching the pair of pointers (2, 4) = (*levels*[0].*pointers*[1], *levels*[0].*pointers*[2]) (lines 5 and 6). Next, we have to find the position ofthe ID 2 among the nodes in the second level. We do this with 3 = *levels*[1].*nodes*.find(2, 4, 2) (line 9). Given that position, we fetch a new pair of pointers (4, 6) = (*levels*[1].*pointers*[3], *levels*[1].*pointers*[4]) (lines 16 and 17). Finally, we know that all completions of the prefix (1, 2) are given by the node IDs found in the range *levels*[2].*nodes*[4, 6), that are 0 and 1. These will be returned by the iterator object created in line 20 of the pseudo code.

We now discuss the time complexity of a triple selection pattern. We use the following nomenclature: n indicates the total number of triples; matches indicates the number of matches for a given pattern; $|S|$, $|P|$ and $|O|$ indicate the number of distinct subjects, predicates and objects respectively. Given an integer sequence S, we assume that: (1) we can randomly access any position of S and retrieve the integer at such position in $O(1)$ time; (2) the complexity of instantiating an iterator over the range $S[i, j)$ and returning every integer in such range is $\Theta(j - i + 1)$, that is linear in the size of the range. Therefore it follows that the S.find(i, j, x) operation can be implemented using binary search in $O(1 + \log(j - i))$ that is $O(1 + \log|S|)$ time for any x and interval. It is then straight forward to see that the pattern ??? is supported in $\Theta(n)$ time, that is $\Theta(1)$ per triple. The patterns with two wildcard symbols are supported in $\Theta(1 + \text{matches})$ time. The patterns with one wildcard need one find operation to be resolved in the second level of the trie dedicated to their support. Therefore, SP? takes $O(1 + \log|P| + \text{matches})$ time, S?O and ?PO take $O(1 + \log|S| + \text{matches})$ and $O(1 + \log|O| + \text{matches})$ time respectively. Finally, the pattern SPO needs two find operations, thus taking $O(1 + \log|P| + \log|O|)$ time.

Supporting range queries. Another relevant characteristicof the introduced layout is that it can support also *range queries*, i.e., queries filtering the set of triples to be returned by means of range constraints. For example, we could impose a limit on the objects of the pattern ?PO by requesting all subjects with a given property *and* having their objects o such that $l < o < r$, with l and r being two fixed values.

In order to support such queries we can modify the default lexicographic assignment of URIs to IDs as follows. Strings still follows a lexicographic ID assignment, i.e.,

```
1  select(triple)
2      i = triple.first
3      if i > levels[0].pointers.size() :          d out of bounds
4          return invalid_iterator()
5      begin = levels[0].pointers[i]
6      end = levels[0].pointers[i + 1]
7      j = begin
8      if triple.second != ? :          d not a wildcard symbol
9          j = levels[1].nodes.find(begin, end, triple.second)
10         if j == −1 :          d j is not found in [begin, end)
11             return invalid_iterator()
12     second = iterator(levels[0].pointers.iterator_at(i),
                         levels[1].nodes.iterator_at(j))
13     i = j
14     if i > levels[1].pointers.size() :
15         return invalid_iterator()
16     begin = levels[1].pointers[i]
17     end = levels[1].pointers[i + 1]
18     j = begin
19     third = iterator(levels[1].pointers.iterator_at(i),
                        levels[2].nodes.iterator_at(j))
20     return iterator(triple.first, second, third)
```

Fig. 2. Function select solving triple selection patterns with one or two wildcard symbols. The input is a *triple* object, assumed to be formed byits *first*, *second* and *third* attributes.

these are sorted lexicographically and consecutive IDs are assigned in such order. Numeric types, instead – the ones over which we could possibly express a range restriction – such as integers or real numbers, dates, and so on[5], are sorted in increasing order and compressed in a distinct data structure, say R. This R data structure is just a sorted integer sequence that, as we are going to illustrate next, supports binary search directly over the compressed representation.

Now, the range restriction $l <$ `?value` $< r$ will be handled as follows.

1) *The lower and upper bounds, l and r, are searched in R to obtain their IDs, say id_A and id_r. In the case l (r) is not present in R, let id_A (id_r) be the ID of the closest value in R smaller than l (larger than r).*
2) *All entities whose IDs are larger than id_A and less than id_r will bound to the variable* `?value` *and returned using the algorithm described in Fig. 2.*

In conclusion, range queries need *at most two* additional searches into a separate data structure with respect to a select query. As we will see in Section 4, the space cost of$R$ is small because its data is sorted and very compressible. However, we do not focus much on range queries in the rest of the paper, hence we assume a traditional lexicographic ID assignment in the following.

Representation. A key characteristic of the trie data structure is to conceptually replace runs of the same ID x in a sequence with the pair (x, *pointer*), where the *pointer* information indicates the run length and where the run is located in the sequence. This already produces significant space savings when triples share many repeated IDs, as it holds for large RDF datasets. Again, refer to Fig. 1 for a basic

Conclusions

In this work we have proposed compressed indexes for the storage and search of large RDF datasets, delivering a remarkably improved effectiveness/efficiency trade-off against existing solutions. for the storage and search of large RDF datasets, delivering a remarkably improved effectiveness/efficiency trade-off against existing solutions. In particular, the extensive ex- perimentation provided has shown that our best trade-off configuration reduces storage requirements by 30 – 60%

References

[1] T. Berners-Lee, J. Hendler, and O. Lassila, "The semantic web," *Scientific American*, vol. 284, no. 5, pp. 34–43, 2001.

[2] M. Wylot, M. Hauswirth, P. Cudré-Mauroux, and S. Sakr, "Rdf data storage and query processing schemes: A survey," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, p. 84, 2018.

[3] M. T. Özsu, "A survey of rdf data management systems," *Front. Comput. Sci.*, vol. 10, no. 3, pp. 418–432, Jun. 2016.

[4] T. Neumann and G. Weikum, "The rdf-3x engine for scalable management of rdf data," *The VLDB JournalThe International Journal on Very Large Data Bases*, vol. 19, no. 1, pp. 91–113, 2010.

[5] ——, "x-rdf-3x: fast querying, high update rates, and consistency for rdf databases," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 256–263, 2010.

[6] P. Yuan, P. Liu, B. Wu, H. Jin, W. Zhang, and L. Liu, "Triplebit: a fast and compact system for large scale rdf data," *Proceedings of the VLDB Endowment*, vol. 6, no. 7, pp. 517–528, 2013.

[7] H. Paulheim and C. Bizer, "Type inference on noisy rdf data," in *International semantic web conference*. Springer, 2013, pp. 510–525.

[8] J. Subercaze, C. Gravier, J. Chevalier, and F. Laforest, "Inferray: fast in-memory rdf inference," *Proceedings of the VLDB Endowment*, vol. 9, no. 6, pp. 468–479, 2016.

[9] C. Weiss, P. Karras, and A. Bernstein, "Hexastore: sextuple indexing for semantic web data management," *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 1008–1019, 2008.

[10] M. A. Martínez-Prieto, M. A. Gallego, and J. D. Fernández, "Exchange and consumption of huge rdf data," in *Extended Semantic Web Conference*. Springer, 2012, pp. 437–452.

[11] M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler, "Matrix bit loaded: a scalable lightweight join query processor for rdf data," in *Proceedings of the 19th international conference on World wide web*. ACM, 2010, pp. 41–50.

[12] D. J. Abadi, A. Marcus, S. Madden, and K. Hollenbach, "Sw-store: a vertically partitioned DBMS for semantic web data management," *VLDB J.*, vol. 18, no. 2, pp. 385–406, 2009.

[13] S. Sankar, M. Singh, A. Sayed, and J. A. Bani-Younis, "An efficient and scalable rdf indexing strategy based on b-hashed-bitmap algorithm using cuda," *International Journal of Computer Applications*, vol. 104, no. 7, 2014.

[14] N. R. Brisaboa, S. Ladra, and G. Navarro, "k^2-trees for compact web graph representation," in *International Symposium on String Processing and Information Retrieval*. Springer, 2009, pp. 18–30.

[15] K. Sadakane, "New text indexing functionalities of the compressed suffix arrays," *Journal of Algorithms*, vol. 48, no. 2, pp. 294–313, 2003.

[16] S. Álvarez-García, N. Brisaboa, J. D. Fernández, M. A. Martínez-Prieto, and G. Navarro, "Compressed vertical partitioning for efficient rdf management," *Knowledge and Information Systems*, vol. 44, no. 2, pp. 439–474, 2015.

[17] N. R. Brisaboa, A. Cerdeira-Pena, A. Farina, and G. Navarro, "A compact rdf store using suffix arrays," in *International Symposium on String Processing and Information Retrieval*. Springer, 2015, pp.