Compressed Indexes for
Fast Search of Semantic Data

Raffaele Perego – Giulio Ermanno Pibiri – Rossano Venturini

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. The extensive experimental analysis, conducted over a wide range of publicly available real-world datasets, reveals 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 $2 - 81\times$.

Index Terms—Indexing; compression; efficiency; RDF

✦

1 INTRODUCTION

THE 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 relationships in order to improve the quality of the data and discover possible inconsistencies [7], [8].

In this work, we focus on the *triple indexing problem* that is to design a static index for the integer triples that attains to efficient resolution of all possible selection patterns using as little space as possible. This is crucial to guarantee practical SPARQL 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 too much 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

- *Raffaele Perego and Giulio Ermanno Pibiri are affiliated to the High Performance Computing Lab, ISTI-CNR, Area della Ricerca di Pisa, Via Giuseppe Moruzzi, 1, 56126 Pisa, Italy.*
  E-mail: {raffaele.perego, giulio.ermanno.pibiri}@isti.cnr.it
- *Rossano Venturini is affiliated to the Department of Computer Science, University of Pisa, Viale Largo Bruno Pontecorvo 3, 56127 Pisa, Italy.*
  E-mail: rossano.venturini@unipi.it
- *This work was partially supported by the BIGDATAGRAPES project (grant agreement #780751, European Union's Horizon 2020 research and innovation programme) and MIUR-PRIN 2017 "Algorithms, Data Structures and Combinatorics for Machine Learning".*

1. https://www.w3.org/RDF
2. http://ogp.me
3. https://www.w3.org/TR/sparql11-overview

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

Our contributions. In particular, our detailed contributions are as follows.

1) We 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 $2 - 4\times$ 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 $2 - 81\times$.

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

2 RELATED WORK

In the last decade many researchers investigated RDF management systems from different perspectives and a complete review of the efforts in this field is out of the scope of this paper. Readers interested in the general topic of RDF data management can refer to two very recent surveys [2], [3]. In the following we focus on the works proposing indexing structures built over the set of triples to support efficient RDF query processing. These works usually exploit string dictionaries to compactly code the URIs occurring in the triples with unique integer IDs and adopt specific domain-dependent techniques to index various S-P-O permutations in order to enhance locality in the access to all the triples matching a given selection pattern: SPO, OSP, SOP, OPS, POS and PSO. Depending on the solution proposed, some or all these permutations are materialized and sorted by the values of the IDs in the three columns.

A simple incarnation of this approach is called *vertical-partitioning* [12], where the permutation PSO is materialized. In particular, a 2-column table is built for each predi-

cate and it lists all the (*subject*, *object*) pairs sorted on the *subject* component to permit fast search and good compression effectiveness. Vertical-partitioning can be generalized to any permutation of the S-P-O components. For example, instead of materializing PSO, the triples can be partitioned by the *subject* component and the (*predicate*, *object*) pairs stored in sorted tables. In the extreme case, all the six possible permutations can be materialized.

HexaStore [9] and RDF-3X [4], [5] follow this exhaustive indexing strategy. They both build and materialize six indexes, one for each possible permutation of the three RDF components. RDF-3X introduces also additional indexes over count-aggregated variants for all three two-dimensional and all three one-dimensional projections. The great flexibility of these solutions at querying time is paid with a very large space occupancy. To partially address this issue, inspired by compression methods for inverted lists in text retrieval systems, RDF-3X exploits VByte to encode the delta gaps computed from the increasingly ordered sequences of integers stored in the clustered B+ trees used for the indexes. The advantage of such organization is that any selection pattern with wildcards possibly occurring in one or two components among the subject, predicate, or object can be efficiently processed on the most suitable index, e.g., the one where the matching triples are stored contiguously in memory. Moreover, also joins for processing structural SPARQL queries are efficiently supported by fast triple selection over two indexes. However, the clear disadvantage is the additional space occupancy and overhead to store and manage the redundant information.

A completely different approach relies on representing the triples in memory with bitmaps [11], [6]. BitMat [11] encodes the RDF data with a 3D bit-cube, where the three dimensions correspond to S, P and O, respectively. TripleBit [6] encodes triples with a bit-matrix. In this matrix each column represents a distinct triple where only two bits are set to 1 in correspondence of the rows associated with the subject and the object of the triple. Symmetrically, each row corresponds to a distinct entity value occurring as subject or object in the triples uniquely identified by the bits set to 1 in the row. Since the resulting bit matrix is very sparse it is compressed at the column level by simply coding the position of the two rows corresponding to the bits set, i.e., the identifiers of the subject and object of the associated triple. The experimental assessment discussed in the paper shows that TripleBit outperforms RDF-3X and BitMat by up to $2 - 4$ orders of magnitude on large RDF datasets. Some work also investigated the impact of graph processing units (GPUs) to process binary matrices [13]. Nevertheless, the space occupancy and scalability of such techniques is not very good.

Other authors have explored how variations of well-known data structures like k^2-trees [14] and the Sadakane's *compressed suffix array* (CSA) [15] can be exploited to compactly represent RDF datasets. In particular, k^2-TRIPLES [16] partitions the dataset into disjoint subsets of (*subject*, *object*) pairs, one per predicate, and represents the (highly) sparse bit matrices with k^2-trees. Another approach called RDFCSA [17], [18] builds an integer CSA index [19] over the sequence of concatenated triple IDs, with the use of truncated Huffman codes on integer gaps and run lengths

for optimized performance. The experimental assessment shows that RDFCSA requires roughly twice the space of the k^2-TRIPLES but it is up to two order of magnitude faster than the former. Although both these solutions outperform existing solutions like RDF-3X in both space usage and query efficiency, no implementation is publicly available to reproduce their results[4].

The RDF index most similar to our solution is HDT-FoQ (*Focused on Querying*) [20], [10], a trie-based solution that exploits the skewed structure of RDF graphs to reduce space occupancy while supporting fast querying. The HDT-FoQ format includes a *header*, detailing logical and physical metadata, the *dictionary*, encoding all the unique entities occurring in the triples as integers, and the set of *triples* encoded in a single SPO trie data structure. In order to support predicate-based retrieval, the second level of the trie is represented with a *wavelet tree* [21]. Finally, additional inverted lists are maintained for object-based triple retrieval. In particular, for each object o, an inverted list is built, listing all pairs (*subject*, *predicate*) of the triples that contain o. Thus, searches are carried out by accessing each pair and searching for it in the trie. A similar approach based on wavelet trees was also proposed by Curé et al. [22].

As we are going to detail next, our base solution is similarly based on the trie data structure but instead of maintaining a single trie we exploit efficient and effective compression strategies resulting in a faster solution for all the triple selection patterns and a smaller space occupancy.

3 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.

3.1 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,
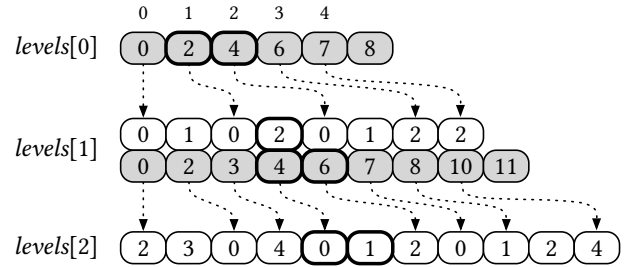
4. Personal communication.

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 avoid permuting back each returned triple.

Each permutation of the triples is represented as a 3-level trie data structure, with nodes at the same level concatenated together to form an integer sequence. We keep track of where groups of siblings begin and end in the concatenated sequence of nodes by storing such pointers as absolute positions in the sequence of nodes. Therefore, the pointers are integer sequences as well. Moreover, since the triples are represented by the trie data structure in sorted order, the n node IDs in the first level of each trie are always complete sequences of integers ranging from 0 to $n-1$ and, thus, can be omitted. We can model each trie data structure with an array $levels[0, 1, 2]$ of three objects, each one having two integer sequences of *nodes* and *pointers*. An exception is represented by $levels[0]$ for which, as discussed above, nodes are missing, and by $levels[2]$ for which pointers are missing.

Refer to Fig. 1 for a pictorial example in which the following 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 in the 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 $S.\text{find}(i, j, x)$ 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 $S.\text{iterator\_at}(i)$ instead instantiates an iterator starting at $S[i]$. We assume that invalid_iterator is a 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 of the ID 2 among the nodes in the second level. We do this with $3 = levels[1].\text{nodes.find}(2, 4, 2)$ (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 $S.\text{find}(i, j, x)$ 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 characteristic of 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 $\ell < o < r$, 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() :        ▷ 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 != ? :        ▷ not a wildcard symbol
9           j = levels[1].nodes.find(begin, end, triple.second)
10          if j == -1 :        ▷ 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 by its *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 $\ell < ?\text{value} < r$ will be handled as follows.

1) *The lower and upper bounds, ℓ and r, are searched in R to obtain their IDs, say id_ℓ and id_r. In the case ℓ (r) is not present in R, let id_ℓ (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_ℓ and less than id_r will bound to the variable $?\text{value}$ 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

5. https://www.w3.org/TR/rdf-sparql-query/#operandDataTypes.

TABLE 1
Performance of various compressors applied to the sequences of *nodes* of all tries and levels for the DBpedia dataset. Performance is reported in bits/triple and in nanoseconds per integer when performing access, find and scan operations. In parentheses we report the percentage of space occupancy taken out by the specific sequence with respect to the whole index.

		SPO				POS				OSP			
		bits/triple	access	find	scan	bits/triple	access	find	scan	bits/triple	access	find	scan
Level 2	Compact	4.74 (4.41%)	1.6	22	3	10.42 (9.69%)	1.4	285	2	22.15 (20.61%)	2.2	63	4
	EF	2.15 (2.33%)	24.4	74	4	4.54 (4.93%)	18.6	533	3	21.37 (23.19%)	34.0	168	5
	PEF	2.01 (2.51%)	65.9	83	5	1.34 (1.67%)	73.1	200	4	18.06 (22.58%)	79.8	135	6
	VByte+SIMD	3.79 (4.07%)	68.2	77	2	3.42 (3.67%)	154.3	13067	1	20.40 (21.90%)	429.3	905	5
Level 3	Compact	27.00 (25.12%)	1.6	31	4	25.00 (23.26%)	2.6	69	4	11.00 (10.24%)	2.2	8	5
	EF	26.90 (29.19%)	31.1	97	6	24.17 (26.23%)	41.6	185	5	5.86 (6.36%)	30.4	60	6
	PEF	26.36 (32.95%)	77.0	115	6	20.31 (25.39%)	90.0	150	6	4.76 (5.95%)	67.5	97	7
	VByte+SIMD	26.57 (28.52%)	418.8	421	6	22.95 (24.64%)	431.6	910	5	8.87 (9.52%)	71.8	75	5

graphical evidence of this fact. We note that the sequences of *nodes* and *pointers* on each trie level can be effectively compressed using a wealth of available techniques, developed to compactly represent (long) integer sequences with excellent search capabilities, e.g., the ones appearing in inverted indexes [24]. A review and discussion of all such techniques is out of scope for the contents of this paper: we briefly overview the ones relevant for our purposes and point the interested reader to the more general survey by Zobel and Moffat [25] and the ones by Pibiri and Venturini [26], [24] for a complete and more detailed description.

Therefore, the problem we face now is the one of compactly representing the levels of each trie by achieving, at the same time, an efficient execution of **select** algorithm that, besides the **find** function, demands fast random **access** to the pointer sequences (see Fig. 2). Below we discuss some different options to achieve this goal, chosen as representative of the many available.

A first (obvious) option would be to minimize the number of bits needed to encode an integer, taking $\lceil \log_2 max \rceil$ bits per value with *max* being the maximum one in the sequence. The advantage of this technique – indicated as **Compact** in the following – lies in its simplicity and efficiency, given that a few bit-wise operations are needed to implement the random **access** operation and, consequently, the **find**(i, j, x) operation that can be supported by binary searching the range $S[i, j]$.

Instead, an elegant integer encoder fulfilling the **select** requirements is *Elias-Fano* (EF) [27], [28]. Elias-Fano combines a fast, namely constant-time, random **access** algorithm with space close to the information theoretic minimum. In particular, the number of bits needed to represent a sorted integer sequence $S[0, n)$ is at most $n\lceil \log(m/n) \rceil + 2n$, where $m \geq S[n-1]$ is the universe of representation of the sequence. The *partitioned* variant (PEF), introduced by Ottaviano and Venturini [29], reduces the space of Elias-Fano by breaking a sequence into partitions and encoding each partition separately. We test this encoder as (one of the) representative of the best space/time trade-off in the literature. Not surprisingly, the adoption of this encoder to model the levels of a trie has been recently used to reduce the space of massive n-gram datasets by a factor of 2 over the most compact trie representation and to provide state-of-the-art time performance [30], [31].

Other approaches focus on representing a sequence of d-gaps, i.e., differences between successive integers, with the requirement of computing a prefix-sum when decoding the d-gapped stream. A common way of speeding up the operations on the compressed sequence, is to divide the sequence into fixed-size blocks, e.g., 128 integers. As a meaningful representative of such technique, we use Variable-Byte (VByte) [32] that achieves the highest decompression speed in the literature, especially when combined with the SIMD-ized decoding algorithm devised by Plaisance et al. [33]. Variable-Byte compresses an integer using the minimum number of bytes needed by its binary representation. The binary representation is divided into chunks of 7 *data* bits with an additional *control* bit inserted to indicate whether the byte is the last byte of the integer. It follows that its simple and byte-aligned nature strongly favours decoding efficiency at the expense of compression effectiveness.

Performance. Table 1 reports the performance of the mentioned techniques when applied to represent the node ID sequences of the tries of DBpedia. The space is expressed as bits per triple (henceforth, bits/triple), whereas the efficiency of the operations access, find and scan in nanoseconds per integer. The reported timings resulted from the average across five runs of the same experiment to smooth fluctuations during measurements.

In order to operate in a meaningful benchmarking setting for the select algorithm, instead of accessing random positions in the sequences or resolving a find in random intervals, we use a set of 5,000 actual triples randomly extracted from the indexed DBpedia dataset. Given a triple (s, p, o) belonging to the trie SPO, we report in the table the time required to access the second level at the position of p (pre-calculated); similarly, we measure the time needed for finding the position of p among the children of s (find). The time reported for scan indicates the time spent per node, when decoding the level sequentially. The same holds for the third level of SPO, as well as for the other tries POS and OSP. The details of the used test machine are reported at the beginning of Section 4.

Now, by looking at the results reported in Table 1, we can express the following considerations.

- All the tested techniques require about the same space for representing the sequences of subjects and objects, with PEF being the most space-efficient.

- There is only a little difference between the space of PEF and Compact when representing the objects of SPO (third level). The reason for this behavior is the *monotonicity* of

the sequences required by the Elias-Fano encoder. In fact, while the sequences of pointers are monotone by construction and, thus, are immediately encodable, the same does not hold for node ID sequences where only sub-sequences of sibling nodes are ordered. Node ID sequences are thus formed by *sorted sub-sequences* with the last element of a sub-sequence being not necessarily smaller than the first one of the next one. In order to encode such sequences with a compressor like Elias-Fano we need to apply a simple transformation adding to the value of each node ID the value of the prefix-sum of the previously coded sub-sequence. This transformation makes Elias-Fano perform poorly whenever the prefix-sum is updated too frequently as it happens in the presence of many small ranges, because the universe of representation grows (very) quickly, thus enlarging the space of representation. As we will show later, most levels of the tries are indeed populated by such tiny ranges, meaning that each node has only few children on average.

- PEF is roughly $2\times$ smaller than the other options on the sequences of predicates.

- The Compact representation is, in general, the fastest for all operations, especially when performing random access that requires $1.4 - 2.6$ nanoseconds. However, it is the least space-efficient as well.

- The PEF variant imposes a penalty of roughly $2\times$ over its un-partitioned counterpart when performing random access, but results to be faster on average for the find operation because it operates inside a partition that is much smaller than the whole sequence. As expected, VByte codes spend more time for find because of the (expensive) decompression of blocks that happens to be competitive only when decoding the sequences of predicates.

For the reasons discussed above, we adopt the following design choices. We use PEF to represent all the sequences of node IDs, except for the last of the trie SPO where we adopt the Compact representation. The pointer sequences are represented using plain EF (not partitioned) given its superior random access efficiency.

From now on, we refer to this solution as the 3T index.

Space breakdowns. Lastly in this subsection, we discuss the space breakdowns reported in parentheses in Table 1. These values indicate the percentage of space out of the whole index required to encode with a given compressor the specific sequence of IDs (the space for pointers is excluded from the count).

As a general consideration, we observe that each permutation takes roughly $1/3$ of the space of the whole index, with the SPO trie being a little larger than the other two. For example, by considering the values for the PEF encoder, we have that the levels of SPO require $2.51\% + 32.95\% = 35.46\%$ of the whole index, whereas the ones of POS 27.06%. Summing up all the values for PEF, we get a total percentage of 91.05%, thus the space for the pointer sequences takes less than 9% of the whole index.

The levels of the index that contribute the most to the overall index space are: the third levels of SPO and POS; the second level of OSP. Specifically, we observe that such levels take practically the *whole* space of the single tries which
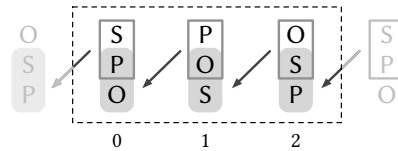

Fig. 3. Graphical representation of our cross-compression technique, in which an arrow $(X)_i \rightarrow (Y)_j$ indicates that level X of trie i is used to cross-compress the level Y of trie $j = (i + 2)$ mod 3, for $i = 0, 1, 2$.

they belong to because the contribution of the other two levels is marginal (e.g., between 5% and 7.5% for PEF). The first levels occupy almost no space because they comprise only the pointers. The second levels of SPO and POS have a small memory footprint for different reasons: for the former, because the number of bits needed to represent a predicate is small compared to the number of bits necessary to represent subjects and objects (see Table 3); for the latter, because the high associativity of predicates partitions the objects' space into a few but very long sorted sequences that are highly compressible. For example, only 1.34 bits/triple are needed for the objects to be represented in the second level of the trie POS. The former reason also causes the third level of OSP to require a small space.

These considerations suggest that the efforts in trying to reduce the space of our RDF index should be mainly targeted to reduce the space for encoding level three of SPO and POS and level two of OSP. This will be the objective of the next section.

3.2 Cross compression

The index layout that we have introduced represents the triples three times in different (cyclic) permutations in order to optimally solve all triple selection patterns. However, the current description does not take advantage of the fact that in this way the same set of triples is represented multiple times and that, consequently, the index has an abundance of redundant information. In this section we develop a novel compression framework that does not compress each trie independently but employs levels of the tries to compress levels of other tries, thus holistically *cross-compressing* the different permutations.

The compression framework is graphically illustrated in Fig. 3. In the picture, we depict the three different permutations and highlight that the levels enclosed in the squared boxes are used to cross-compress the (lower) levels enclosed by the shaded boxes as pointed to by the arrows. Cross-compression works by noting this crucial property: *the nodes belonging to a sub-tree rooted in the second level of trie j are a subset of the nodes belonging to a sub-tree rooted in the first level of trie i, with $j = (i + 2)$ mod 3, for $i = 0, 1, 2$.* The correctness of this property follows automatically by taking into account that the triples indexed by each permutation are the same. Therefore, the children of x in the second level of trie j can be re-written as the *positions* they take in the (larger, enclosing) set of children of x in the first level of trie i. Re-writing the node IDs as positions relative to the set of children of a sub-tree yields a clear space optimization because the number of children of a given node is much

TABLE 2
Number of children of the trie nodes for DBpedia.

Trie	Level	Average	Maximum
SPO	1	5.54	52
	2	2.32	8,489
POS	1	91,578.32	21,219,244
	2	2.59	10,141,311
OSP	1	2.70	10,141,327
	2	1.13	10

```
1  map(parent, child)
2     begin = levels[0].pointers[parent]
3     end = levels[0].pointers[parent + 1]
4     return levels[1].nodes.find(begin, end, child) − begin
```

```
1  unmap(parent, child)
2     begin = levels[0].pointers[parent]
3     end = levels[0].pointers[parent + 1]
4     return levels[1].nodes[child + begin]
```

Fig. 4. Functions used to map and unmap a child ID *conditionally* to its sibling IDs.

smaller (on average) than the number of distinct subjects or objects. (See also Table 3 for the precise statistics.)

We claim that the average number of children for the nodes in the first levels of the tries is the key statistic affecting the effectiveness of the described cross-compression technique because smaller numbers require a smaller number of bits to be represented. We report this statistic in Table 2 for the DBpedia dataset. The data reported in the table is self-explanatory: except for the first level of POS, the average number of children is *very* low, being actually less than 3 for OSP, across the different permutations. As already mentioned, the high associativity of the predicates causes each predicate to have many children, i.e., several orders of magnitude more than the children of subjects and objects. We also show the maximum number of children, i.e., a maximum of m indicates that there is a least one node that has m children. While this value can be very distant from the average, for some levels like the second of SPO and OSP is actually similar, meaning that the selection patterns SP? and S?O have a robust worst-case guarantee.

Fig. 4 shows a compact pseudo code illustrating how a *child* ID can be rewritten conditionally to its *parent* ID in the described cross-compression framework. Conversely, the function unmap shows what actions are needed to recover the original ID before returning the result to the user. In particular, this latter function needs to be applied to the third component of *every* triple matched by a selection pattern, hence partially eroding the retrieval efficiency. Note that fast random access is needed by the unmap function, thus it could be convenient to model *levels*[1].*nodes* with a Compact representation (see also Table 1 for benchmark numbers).

Discussion. We have seen that the nodes in the third level of the POS trie can be cross-compressed using the sub-trees rooted in the first level of the trie OSP, whose average number of children is actually less than 3 on real-world datasets. Therefore, what we should expect is to have the third level of POS mostly populated with values in $\{0, 1, 2\}$ that requires just two bits to be written instead of more than 20 bits to represent each subject. Therefore, we argue that a significant space saving can be achieved by applying this technique, introducing a slowdown for two (out of eight) selection patterns only, i.e., ?PO and ?P? that are solved on the trie POS. As already motivated, the second level of OSP is represented with Compact.

We also argue that cross-compressing the other two permutations, i.e., SPO and OSP, yields only modest advantages. Again, refer to Table 2. To cross-compress the third level of SPO we use the children branching out from the predicates to the objects. But, as already noted, a predicate has many children, thus dwarfing the corresponding reduction in the average number of bits needed to represent a mapped ID. For the permutation OSP, instead, we have to shrink the node ID of the predicates. But these are already encoded compactly, e.g., in less than 4.8 bits/triple on average for DBpedia (see Table 1). Thus, given that we can not expect great space savings in these two cases, we consider the cross-compressed index to be the one with cross-compression on the POS triples and that we indicate with CC in the following.

3.3 Eliminating a permutation

The low number of predicates exhibited by RDF data in combination to the corresponding very few children of the nodes in the second level of the trie OSP, leads us to consider a different select algorithm for the resolution of the query pattern S?O, able to take advantage of such skewed distribution of the predicates. In fact, recall from Table 2 that on the DBpedia dataset, for a given (*subject*, *object*) pair, just 1.13 predicates are returned on average and 10 *at most*. Also recall from Section 3.1 that the pattern S?O is solved on the trie OSP trie with the select algorithm in Fig. 2 by performing a find operation followed by a scan.

An alternative approach for solving S?O. The idea is to pattern match S?O directly over the SPO permutation with the algorithm illustrated in Fig. 5. For a given subject s and object o, in short, we operate as follows. We consider the set of all the predicates that are children of s. For each predicate p_i in the set, we determine if the object o is a child of p_i with the find function: if it is, then (s, p_i, o) is a triple to return. We refer to such strategy as the enumerate algorithm, whose correctness is immediate.

As partially motivated, we argue that the efficiency of the outlined algorithm is due to several distinct facts.

- Although the worst-case time complexity of the algorithm is $O(1 + |P|(1 + \log |O|))$, the small number of predicates as children of a given subject implies that the for loop in Fig. 5 performs few iterations. While these children are few per se, e.g., *at most* 52 for DBpedia, the iterations will be on average far less. For example, less than 6 iterations are needed on average for DBpedia. (See detailed statistics in Table 2.)

- For a given (subject, predicate) pair, we have a very limited number of children to be searched for o, thus making the find function run *faster* by *short scans* rather than via binary

```
1  enumerate(s, o)
2      begin = levels[0].pointers[s]
3      end = levels[0].pointers[s + 1]
4      for i = begin; i != end; i = i + 1 :
5          p = levels[1].nodes[i]
6          j = levels[1].pointers[i]
7          k = levels[1].pointers[i + 1]
8          position = levels[2].nodes.find(j, k, o)
9          if position != −1 :            ▷ o is found in [j, k]
10             (s, p, o) is a matching triple
```

Fig. 5. The algorithm specialized to pattern match S?O.

search. On DBpedia, an average of 2.32 children have to be scanned per pair.

- Furthermore, the enumerate algorithm operates on the SPO permutation whereas the select algorithm on OSP: but if we consider the rows for the PEF encoder in Table 1, we see that a find operation issued on the second level of OSP costs 1.6 times more the amount of nanoseconds spent on SPO. Thus, avoiding percolating the trie OSP produces further savings.

Discussion. In the light of devising a competitive algorithm to pattern match S?O on SPO, we consider another index layout. Combining the design introduced in Section 3.1 with the present considerations, *five* out of the eight different selection patterns can be solved efficiently by the trie SPO, i.e.: SPO, SP?, S??, S?O and ???. In order to support two more selection patterns, we can either chose to: (1) materialize the permutation POS for *predicate-based* retrieval (query patterns ?PO and ?P?); (2) materialize the permutation OPS for *object-based* retrieval (query patterns ?PO and ??O). The choice of which permutation to maintain depends on the statistics of the selection patterns that have to be supported. We stress that the introduced enumerate algorithm allows us to actually *save the space* for a third permutation that, as we have already pointed out in Section 3.1, costs roughly 1/3 of the whole space of the index. We call this solution the 2T index, with two concrete instantiations: 2Tp (predicate-based) and 2To (object-based).

The only selection pattern that is not immediately supported is ?P? for 2To (symmetrically, ??O for 2Tp). However, we know that predicates are very associative, thus the list of subject nodes having a predicate parent is, on average, very long and highly compressible. Therefore, we can afford to maintain a two-level structure PS that, for every predicate p, maintains the list of all subjects that appear in triples having predicate p. Recall from our discussion concerning the space breakdowns in Section 3.1 that the similar structure PO costs just 1.34 bits/triple for DBpedia, thus PS should be only a little more costly given that the SP pairs are more in number than the PO pairs (see Table 3). Therefore, we proceed as follows.

1) *Access the list* $p \to [s_1, \ldots, s_n]$.
2) *For* $i = 1, \ldots, n$, *consider the pair* (s_i, p) *and pattern match* $s_i p$? *against the permutation* SPO.

Similarly for the index 2Tp, if we consider a query pattern ??O with specified object o, |P| find operations are needed

to locate all the occurrences of o in the second level of the trie POS.

1) *For every predicate p, consider its children.*
2) *Determine if o is among them with the* find *function. If yes, then return all triples (s, p, o) with s being a child of the pair (p, o).*

Again, the correctness of these two algorithms is immediate. Their worst-case time complexities are respectively $O(|S|(1 + \log |P|) + \text{matches})$ for ?P? and $O(|P|(1 + \log |O|) + \text{matches})$ for ??O. In order to distinguish them from select and enumerate, in the following analysis we refer to both approaches as inverted.

4 EXPERIMENTAL ANALYSIS

In this section we first compare the performance of the three solutions that we have introduced in the previous section, i.e., 3T (Section 3.1), 3T with *cross compression*, indicated as CC (Section 3.2) and 2T (Section 3.3). Then, we compare against the competitive approaches at the state-of-the-art that we have discussed in Section 2.

Datasets. We perform our experimental analysis on the following standard datasets, whose statistics are summarized in Table 3. DBLP [34] is the dump of the DBLP Computer Science Bibliography collected during 2017. Geonames [35] is the official 2012 dump collected by geonames.org. DBpedia [23] is the English version 3.9 of DBpedia, dubbed as "the nucleus for a Web of Data". WatDiv [36] is the "1B Triples" Waterloo SPARQL Diversity Test Suite dataset, available for download at https://dsg.uwaterloo.ca/watdiv (Section 4). LUBM [37] is a synthetic dataset generated using the methodology described by Lehigh University Benchmark, that is an established and widely-used benchmark for semantic data. (In particular, the dataset was generated using the tools from https://github.com/rvesse/lubm-uba with option -u 10000.) Freebase [38] is the last available data dump of Google Freebase collected in December 2013.

Experimental setting and methodology. All the experiments are performed on a server machine with 4 Intel i7-7700 cores (@3.6 GHz), with 64 GB of RAM DDR3 (@2.133 GHz), running Linux 4.4.0, 64 bits. We implemented the indexes in C++14 and compiled the code with gcc 7.3.0 using the highest optimization setting, i.e., with compilation flags -O3 and -march=native.

The indexes are saved to the disk after construction and loaded in internal memory to be queried. In order to avoid disk caching effects, we clear the cache of the disk before measuring query timings. To measure the query processing speed, we use a set of 5,000 triples drawn at random from the datasets and set 0, 1 or 2 wildcard symbols. The reported results are averages over five runs of queries in order to smooth fluctuations during measurements. All queries run on a single processing core.

In all tables, speed up factors are taken with respect to the values highlighted in bold font. Notation x (+p%) means that if we subtract p% of the space of x, we obtain the value in bold font, say z. In other words: x takes p% more space than z.

TABLE 3
Datasets basic statistics, reporting the total number of triples, number of distinct subjects (S),
distinct predicates (P), distinct objects (O) and distinct pairs.

Dataset	Triples	S	P	O	SP pairs	PO pairs	OS pairs
DBLP	88,150,324	5,125,936	27	36,413,780	58,476,283	46,468,249	70,234,083
Geonames	123,020,821	8,345,450	26	42,728,317	118,410,418	45,096,877	112,961,698
DBpedia	351,592,624	27,318,781	1,480	115,872,941	151,464,424	135,673,814	311,567,728
WatDiv	1,092,155,948	52,120,385	86	92,220,397	230,085,646	111,561,465	1,092,137,931
LUBM	1,334,681,190	217,006,852	17	161,413,040	1,060,824,925	195,085,216	1,334,459,593
Freebase	2,067,068,154	102,001,451	770,415	438,832,462	878,472,435	722,280,094	1,765,877,943

4.1 The permuted trie index

To assess the performance of our solutions we chose to show the results for the *real-world* datasets of DBLP, Geonames, DBpedia and Freebase, given that similar trends and conclusions were observed on the others. We will use the other two (synthetic) datasets, WatDiv and LUBM, to test the speed of selection patterns occurring in the corresponding SPARQL query logs in Section 4.2.

Compression effectiveness. We first comment on the compression effectiveness achieved by our different solutions. Refer to the upper part of Table 4, where we report the average number of bits spent per represented triple on the different datasets. We sorted the rows in order of increasing effectiveness, thus we can immediately conclude that the most compact index is 2Tp, with CC in between the values of 3T and 2T. The 2Tp variant is even smaller than 2To because it materializes the permutation POS that requires less space than OPS as maintained by 2To. In particular, the compression level exhibited by 3T lies in the range of 72 – 81 bits/triple and as expected, the refinements devised in Section 3.2 and 3.3 pay off. The CC index brings a further saving of 11% on average, because the space of representation of the third level of POS is almost dissolved, being actually reduced by roughly 4× on the different datasets (but the Compact compressor on the second level of OSP takes more space than PEF, as used by 3T). Instead, the 2T indexes reduces the space by 30% on average since they basically eliminate one permutation of the triples, hence optimizing by roughly 1/3.

Triple selection patterns. Now we examine the speed of all the different triple selection patterns by commenting on the lower part of Table 4, where we report the timings in nanoseconds spent per returned triple. In particular, patterns are grouped together when solved by the same trie (or have exactly the same performance). Let us proceed in order, by discussing each group.

- The index component represented by the trie SPO is common by the three solutions, thus the results for SPO, SP?, S??, ??? are the same. We can see that the indexes solve a lookup operation, i.e., the query pattern SPO with all symbols specified, in a fraction of a microsecond, specifically 1/5 – 1/2 μsec. This is basically the time needed for two find operations, respectively on the second and third levels of the trie. Observe how, instead, the "fixed" cost of the find operation on the second level is amortized by the number of matching triples for the pattern SP? on the larger DBpedia and Freebase. Similarly, the time spent per triple by the patterns S?? and ??? is just a handful of

TABLE 4
Comparison between the performance of 3T, CC and 2T indexes,
expressed as the total space in bits/triple
and in average ns/triple for all the different selection patterns.

	Index	DBLP	Geonames	DBpedia	Freebase
		bits/triple	bits/triple	bits/triple	bits/triple
	3T	75.24 (+31%)	71.59 (+32%)	80.64 (+33%)	74.20 (+30%)
	CC	63.54 (+18%)	67.04 (+27%)	66.91 (+19%)	70.46 (+26%)
	2To	56.46 (+8%)	53.23 (+8%)	57.51 (+6%)	55.72 (+6%)
	2Tp	**51.99**	**48.98**	**54.14**	**52.17**
		ns/triple	ns/triple	ns/triple	ns/triple
SPO	*all*	203	221	353	521
SP?	*all*	197	347	11	3
S??	*all*	28	40	10	3
???	*all*	11	13	9	9
S?O	3T,CC	2,490 (5.6×)	3,767 (7.7×)	1,833 (2.6×)	6,547 (1.8×)
	2To,2Tp	**445**	**490**	**692**	**3,736**
?PO	**3T,2To,2Tp**	**5**	**5**	**5**	**5**
	CC	12 (2.4×)	15 (3.0×)	16 (3.2×)	14 (2.8×)
??O	3T,CC	12 (2.4×)	12 (2.4×)	12 (2.4×)	10 (2.0×)
	2To	**5**	**5**	**5**	**5**
	2Tp	5 (1.0×)	5 (1.0×)	6 (1.2×)	10 (2.0×)
?P?	**3T,2Tp**	**9**	**8**	**6**	**6**
	CC	21 (2.3×)	36 (4.5×)	30 (5.0×)	29 (4.8×)
	2To	81 (9.0×)	138 (17.2×)	22 (3.7×)	18 (3.0×)

nanoseconds.

- We now discuss the pattern S?O that is solved by 3T and CC with the select algorithm (Fig. 2) on the trie OSP, but by 2T with the enumerate algorithm (Fig. 5) on the trie SPO. In Section 3.3, we have discussed about the efficiency of the enumerate algorithm and motivated its advantages over the select algorithm. In fact, we observe from Table 4 that enumerate is faster than select by 4.4× on average (a minimum of 1.8× on Freebase, and up to 7.7× on Geonames). In particular, we have claimed that the crucial statistic affecting the performance of the enumerate algorithm is the very low associativity of the subjects, i.e., the limited number of predicates as children of a given subject.

Therefore, in order to validate our hypothesis, we inspect the full spectrum of possibilities in Fig. 7, where we illustrate how the time for S?O changes for queries whose subjects have different number of children. We show the result on DBpedia for space constraints (similar behaviors were observed for the other datasets anyway). For DBpedia the number of children of a subject, henceforth indicated with C, varies between 1 and 52 (see also Table 2). As it is clear from the plot, whenever $C < 30$ the enumerate algorithm is faster and looses only for higher values, because of the cache-friendly behaviour of select due to its scan-based
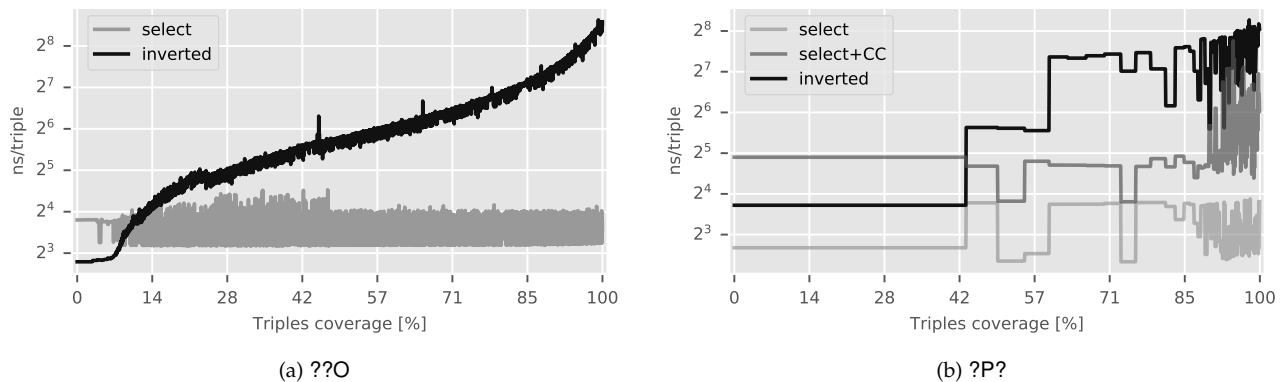
(a) ??O

(b) ?P?

Fig. 6. Average ns/triple by *decreasing* number of matches, for the query pattern ??O and ?P?.
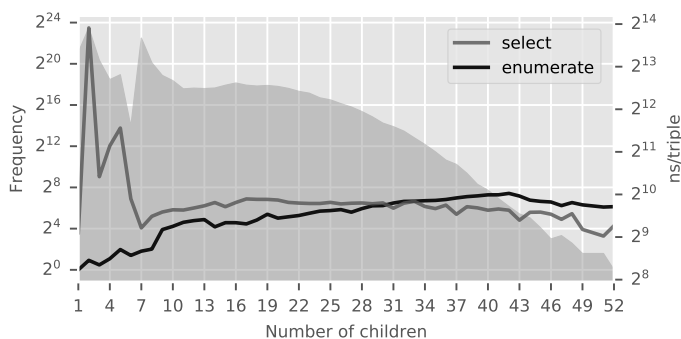

Fig. 7. Comparison between the select and enumerate algorithm when pattern matching S?O on DBpedia, for queries having subjects whose number of children (predicates) is between 1 and 52. On the background, we show the distribution of the number of children.

nature. In particular, on the background of the plot we also show the distribution of C, i.e., how many subjects have $C = c$ children, for $c = 1..52$. The distribution explains that the majority of subjects have only few children, and in correspondence of such values, the enumerate algorithm is *much* faster than select (e.g., up to $43\times$ for $C = 2$), thus explaining why on average is faster as well.

- The pattern ?PO illustrates how the cross-compression affects the retrieval of the subjects on the third level of the POS permutation. For every match, the unmap function must be executed, hence requiring a random access to be performed on the second level of OSP (a potential cache-miss). This results in a slowdown of roughly $3\times$.

- The number of nanoseconds per triple for the pattern ??O is controlled by the average selectivity of the queries: while most of the queries have only a few matches, the presence of low-selective queries maintains the average response time very low. Also, notice that it is solved $2.4\times$ faster by 2To because it traverses the trie OPS and not OSP as it is done by the 3T and CC indexes, thus employing find operations on the sequence of predicates rather than subjects that are faster (see details in Table 1). It is also interesting to note that the pattern is solved efficiently by the 2Tp variant on average, given that the cost of the $|P|$ find operations is well amortized by the number of returned

matches. However, when $|P|$ grows significantly and the results per query are very few, the response time per occurrence is much higher. This *output-sensitive* behaviour is a well-known characteristic of RDF stores. Therefore, for completeness, we also show in Fig. 6a how the time per triple changes by *decreasing* number of matches, until we cover the whole set of triples. The experiment shows that while the select is less output-sensitive, indeed for a reasonably good fraction of the triples, e.g., 25%, the *average* query time of inverted is close to the one of select and even faster for $\approx$10% of the triples.

- Similar considerations hold for the ?P? query pattern, whose corresponding stress behaviour is shown in Fig. 6b: the select offers the best worst-case guarantee across all different output sizes, with the cross-compression technique (select+CC) and the inverted algorithm both imposing overheads due to additional cache-misses. Observe how inverted is actually faster than select+CC for more than 40% of the triples, with the latter being in trade-off position between the former and select.

Range queries. To asses the efficiency of range queries, supported as we explained in Section 3.1, we used the WatDiv dataset that contains several numeric types (primarily integers and dates). We tested query patterns of the form ?P? and ?PO using the 2Tp index. Range constraints are expressed on the object components, hence the queries are handled by the trie POS of 2Tp. We determined an average running time of 4.3 ns/triple. The extra space of the data structure that we use to support such queries is very small as expected – less than 0.1 bits/triple – when compressed using PEF as we did in our implementation. In conclusion, range queries are supported efficiently in both time and space regards.

Discussion. From the above considerations concerning the performance of our solutions, we conclude that: (1) the 3T index is the one delivering best *worst-case* performance guarantee for all triple selection patterns; (2) the 2T variants reduce its space of representation by $25 - 33\%$ *without* affecting or even improving the retrieval efficiency on most triple selection patterns (*only one* pattern has a lower query throughput in the worst-case); (3) the cross-compression technique is outperformed by the 2T index layouts for space

usage but offers a better worst-case performance guarantee than 2Tp for the pattern ?P?. In particular, we have shown that *only one* selection pattern out of the eight possible has a lower query throughput in the worst-case for the 2T indexes (??O on 2Tp; ?P? on 2To).

Therefore, as a reasonable trade-off between space and time concerns, we elect 2Tp as the solution to compare against the state-of-the-art alternatives in the following.

4.2 Overall comparison

In this section, we compare the performance of our selected solution 2Tp against the competitive approaches HDT-FoQ [10] and TripleBit [6] described in Section 2, since these both outperform RDF-3X [5] and BitMat [11].

In particular, the experimental analysis provided by the authors of HDT-FoQ [10] reports that RDF-3X is larger than their own index by $3.8\times$, $4.6\times$ and $3\times$ on DBLP, Geonames and DBpedia respectively, and also slower by a factor of $4-8\times$ on most selection patterns (HDT-FoQ scores worse only for the pattern ?P? because of the penalty induced by representing predicate sequences with wavelet trees, as we are going to confirm next). This is not surprising given that all the six triple permutations, plus aggregate indexes, are materialized by RDF-3X with a severe query processing overhead due to expensive I/O transfers. Similar results are obtained by the authors of TripleBit [6], whose experiments on LUBM show that RDF-3X is $2\times$ larger and $2-14\times$ slower. In the same paper BitMat is shown to be even larger than RDF-3X and up to several orders of magnitude slower[6].

For both HDT-FoQ and TripleBit, we (obviously) do not consider the space needed for the string dictionaries, as well as the time needed for dictionary access/lookup. We use the C++ libraries as provided by the corresponding authors and available at https://github.com/rdfhdt/hdt-cpp and https://github.com/nitingupta910/TripleBit, respectively. Furthermore, both libraries are compiled with the same compiler we used for our own code (gcc 7.3.0) and using the same optimization flags -O3 and -march=native.

Table 5 reports the space of the indexes and the timings for the different selection patterns, but excluding (due to page limit) the ones for SPO and ???: our approach is anyway faster for both by at least a factor of $3\times$ (TripleBit does not support the query pattern SPO). Concerning the space, we see that the 2Tp index is significantly more compact, specifically by 30% and almost 60% compared to HDT-FoQ and TripleBit respectively, on average across all different datasets (TripleBit fails in building the index on Freebase). Concerning the speed of triple selection patterns, most factors of improved efficiency range in the interval $2-5\times$ and, depending on the pattern examined, we report peaks of $26\times$, $49\times$, $81\times$ or even $2,057\times$.

To further confirm the results, we execute all triple selection patterns needed to solve the SPARQL queries from the publicly available query logs of WatDiv[7] and LUBM[8], that are both well-established benchmarks for RDF data.

---

6. To confirm this result, we built the BitMat index on DBpedia, with flag COMPRESS_FOLDED_ARRAY as suggested by the author Medha Atre, and measured a space occupancy of 483.72 bits/triple.

7. http://grid.hust.edu.cn/triplebit/watdiv.txt

8. http://swat.cse.lehigh.edu/projects/lubm/queries-sparql.txt

---

TABLE 5
Comparison between the performance of different indexes, expressed as the total space in bits/triple and in average ns/triple.

Index		DBLP	Geonames	DBpedia	Freebase
		bits/triple	bits/triple	bits/triple	bits/triple
	2Tp	**51.99**	**48.98**	**54.14**	**52.17**
	HDT-FoQ	76.89 (+32%)	88.73 (+45%)	76.66 (+29%)	83.11 (+37%)
	TripleBit	125.10 (+58%)	120.03 (+59%)	130.07 (+58%)	—
		ns/triple	ns/triple	ns/triple	ns/triple
?PO	2Tp	**5**	**5**	**5**	**5**
	HDT-FoQ	12 (2.4×)	13 (2.6×)	14 (2.8×)	13 (2.6×)
	TripleBit	15 (3.0×)	13 (2.6×)	14 (2.8×)	—
S?O	2Tp	**445**	**490**	**692**	**3736**
	HDT-FoQ	1,789 (4.0×)	2,097 (4.3×)	3,010 (4.3×)	0.7×10^7 (2,057×)
	TripleBit	11,872 (26.7×)	13,008 (26.5×)	18,023 (26.0×)	—
SP?	2Tp	**197**	**347**	**11**	**3**
	HDT-FoQ	640 (3.2×)	897 (2.6×)	30 (2.7×)	9 (3.0×)
	TripleBit	1,222 (6.2×)	927 (2.7×)	42 (3.8×)	—
S??	2Tp	**28**	**40**	**10**	**3**
	HDT-FoQ	110 (3.9×)	154 (3.9×)	29 (2.9×)	9 (3.0×)
	TripleBit	2,275 (81.2×)	3,261 (81.5×)	490 (49.0×)	—
?P?	2Tp	**9**	**8**	**6**	**4**
	HDT-FoQ	108 (12.0×)	173 (21.6×)	32 (5.3×)	41 (6.8×)
	TripleBit	28 (3.1×)	28 (3.5×)	40 (6.7×)	—
??O	2Tp	**5**	**5**	**6**	**10**
	HDT-FoQ	17 (3.4×)	17 (3.4×)	18 (3.0×)	18 (1.8×)
	TripleBit	24 (4.8×)	60 (12.0×)	24 (4.0×)	—

TABLE 6
Performance in bits/triple and in average seconds spent by the different solutions to execute the sequence of triple selection patterns generated for the SPARQL queries in the WatDiv and LUBM logs.

Index	WatDiv		LUBM	
	bits/triple	sec/query	bits/triple	sec/query
2Tp	**54.16**	**0.08**	**50.01**	**0.73**
HDT-FoQ	68.79 (+21%)	19.67 (238.2×)	92.41 (+46%)	17.26 (23.7×)
TripleBit	129.49 (+58%)	0.34 (4.1×)	111.93 (+55%)	4.01 (5.5×)

We used the query planning algorithm of TripleBit to obtain a serial decomposition of the SPARQL queries into atomic selection patterns. We argue that this methodology has the advantage of fairly testing the different indexes over the same set of patterns as executed in the same order, thus really testing the raw speed of the underlying indexing data structure that is our goal in this work. The results of the benchmark, illustrated in Table 6, closely match the ones already discussed in Table 5. In particular, our index optimizes the space of representation by $21-58\%$ and is remarkably faster than TripleBit by $4.5\times$ on average and by up to $238\times$ than HDT-FoQ on WatDiv. Most selection patterns for both query logs have ?P? and ?PO forms. In fact, observe how our solution is $3-6.7\times$ faster than TripleBit on these patterns, with HDT-FoQ being the worst because of the cache-misses introduced by representing the predicate sequences with (potentially, tall) wavelet trees.

5 CONCLUSIONS AND FUTURE WORK

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. In particular, the extensive experimentation provided has shown that our best trade-off configuration reduces storage requirements by $30-60\%$

and provides $2 - 81\times$ better efficiency, on real-world RDF datasets with up to 2 billions of triples.

Future work could target the two related problems mentioned in Section 1, that is: (1) providing an efficient *string dictionary* data structure and (2) devising a novel *query planning* algorithm.

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 Journal—The 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. 103–115.

[18] A. Cerdeira-Pena, A. Farina, J. D. Fernández, and M. A. Martínez-Prieto, "Self-indexing rdf archives," in *2016 Data Compression Conference (DCC)*. IEEE, 2016, pp. 526–535.

[19] A. Fariña, N. R. Brisaboa, G. Navarro, F. Claude, Á. S. Places, and E. Rodríguez, "Word-based self-indexes for natural language text," *ACM Transactions on Information Systems (TOIS)*, vol. 30, no. 1, p. 1, 2012.

[20] J. D. Fernández, M. A. Martínez-Prieto, and C. Gutierrez, "Compact representation of large rdf data sets for publishing and exchange," in *International Semantic Web Conference*. Springer, 2010, pp. 193–208.

[21] R. Grossi, A. Gupta, and J. S. Vitter, "High-order entropy-compressed text indexes," in *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, 2003, pp. 841–850.

[22] O. Curé, G. Blin, D. Revuz, and D. C. Faye, "Waterfowl: A compact, self-indexed and inference-enabled immutable RDF store," in *Proceedings of the 11th International Conference on The Semantic Web: Trends and Challenges (ESWC)*, 2014, pp. 302–316.

[23] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives, "Dbpedia: A nucleus for a web of open data," pp. 722–735, 2007.

[24] G. E. Pibiri and R. Venturini, "Techniques for inverted index compression," *CoRR*, vol. abs/1908.10598, 2019. [Online]. Available: http://arxiv.org/abs/1908.10598

[25] J. Zobel and A. Moffat, "Inverted files for text search engines," *ACM computing surveys (CSUR)*, vol. 38, no. 2, p. 6, 2006.

[26] G. E. Pibiri and R. Venturini, "Inverted index compression," *Encyclopedia of Big Data Technologies*, pp. 1–8, 2018.

[27] P. Elias, "Efficient storage and retrieval by content and address of static files," *Journal of the ACM (JACM)*, vol. 21, no. 2, pp. 246–260, 1974.

[28] R. M. Fano, "On the number of bits required to implement an associative memory," *Memorandum 61, Computer Structures Group, MIT*, 1971.

[29] G. Ottaviano and R. Venturini, "Partitioned elias-fano indexes," in *Proceedings of the 37th international ACM SIGIR conference on Research & development in information retrieval*. ACM, 2014, pp. 273–282.

[30] G. E. Pibiri and R. Venturini, "Efficient data structures for massive n-gram datasets," in *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, 2017, pp. 615–624.

[31] ——, "Handling massive n-gram datasets efficiently," *ACM Trans. Inf. Syst.*, vol. 37, no. 2, pp. 25:1–25:41, Feb. 2019.

[32] L. H. Thiel and H. Heaps, "Program design for retrospective searches on large data bases," *Information Storage and Retrieval (ISR)*, vol. 8, no. 1, pp. 1–20, 1972.

[33] J. Plaisance, N. Kurz, and D. Lemire, "Vectorized VByte decoding," in *International Symposium on Web Algorithms (iSWAG)*, 2015.

[34] DBLP, "Computer science bibliography," http://www.rdfhdt.org/datasets, 2017.

[35] geonames.org, "Geonames," http://www.rdfhdt.org/datasets, 2012.

[36] G. Aluç, O. Hartig, M. T. Özsu, and K. Daudjee, "Diversified stress testing of rdf data management systems," in *International Semantic Web Conference*. Springer, 2014, pp. 197–212.

[37] LUBM, "Lehigh university benchmark," http://swat.cse.lehigh.edu/projects/lubm, 2019.

[38] Google, "Freebase data dumps," https://developers.google.com/freebase, 2013.

Raffaele Perego (http://hpc.isti.cnr.it/~raffaele/) is a senior researcher at ISTI-CNR, where he leads the High Performance Computing Lab (http://hpc.isti.cnr.it). His main research interests include large-scale information systems, information retrieval, data mining, and machine learning. He co-authored more than 150 papers on these topics published in journals and proceedings of international conferences.

Giulio Ermanno Pibiri (http://pages.di.unipi.it/pibiri) is a postdoctoral researcher at ISTI-CNR, High Performance Computing Lab. He received a Ph.D. in Computer Science from the University of Pisa in 2019. His research interests involve data compression algorithms for indexing massive datasets, data structures and information retrieval with focus on efficiency.

Rossano Venturini (http://pages.di.unipi.it/rossano) is Associate Professor of Computer Science at the University of Pisa. He received his Ph.D. from the University of Pisa in 2010. His research interests are mainly focused on the design and the analysis of algorithms and data structures with focus on indexing and searching large textual collections. He won two Best Paper Awards at ACM SIGIR in 2014 and 2015.