Compressed Indexes for Fast Search of Semantic Data
(Extended Abstract)

Raffaele Perego
ISTI-CNR, Pisa, Italy

Giulio Ermanno Pibiri
ISTI-CNR, Pisa, Italy

Rossano Venturini
University of Pisa, Pisa, Italy

Abstract—The sheer increase in volume of RDF data demands efficient solutions for the triple indexing problem, that is devising 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. 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 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.

I. THE PERMUTED TRIE INDEX

The Resource Description Framework (RDF) is a W3C standard offering a general graph-based model for describing information as a set of (subject, predicate, object) relations, known as triples. 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 an integer ID to save space, so that each triple in the dataset can be represented with three integers. In this work, we focus on reducing the space of representation of the integer triples while attaining to efficient resolution of all possible selection patterns, noting that this is crucial for guaranteeing practical SPARQL query evaluation [Perego et al., 2020].

As a high-level overview, our solution maintains three different permutations of the triples, with each permutation sorted to allow efficient searches and effective compression as we are going to detail next. 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, 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 trie with 3 levels, where 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. The advantage of this 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, 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 the following, we refer to this solution as the 3T index.

Cross Compression. It is possible to employ levels of the tries to compress levels of other tries, thus holistically cross-compressing the different permutations. 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 smaller (on average) than the number of distinct subjects or objects (see also Table II for the precise statistics).

Eliminating a Permutation. The low number of predicates exhibited by RDF data 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. The idea is to pattern match S?O directly over the SPO permutation. 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 $p$ in the set, we determine if the object $o$ is a child of $p$ with a search operation: if it is, then ($s, p, o$) is a triple to return. In the light of this algorithm, we consider another index layout. In fact, now five of out the eight different selection patterns can be solved efficiently by the trie SPO, i.e.: SPO, SP?, S??, S?O and ??O. In order to support other two 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 ?O and ??O). The choice of which permutation to maintain depends on the statistics of the selection patterns that have to be supported.
indexes. We compiled the code and 2T outperformed by the out of the eight possible has a lower query throughput for a third permutation that costs roughly 1/3 saving the space for a third permutation that costs roughly 1/3 saving the space.

We stress that the introduced algorithm allows us to actually save the space for a third permutation that 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).

## II. Experiments

We perform our experimental analysis on large and publicly available standard datasets, whose statistics are summarized in Table II. All the experiments are performed on a server machine with 4 Intel i7-7700 cores (@3.6 GHz), 64 GB of RAM DDR3 (@2.133 GHz) and running Linux 4.4.0, 64 bits. The C++14 implementation of our indexes is freely available at https://github.com/nitingupta910/TripleBit, respectively. Table III 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× (TripleBit does not support the query pattern SPO). Concerning the speed, we see that the 2T 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× and, depending on the pattern examined, we report peaks of 26×, 49×, 81×, or even 2057×.

### The Permuted Trie Index

By looking at the results reported in Table I, we can 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 out of the eight possible 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 ???. Therefore, as a reasonable trade-off between space and time, we elect 2Tp as the solution to compare against the state-of-the-art alternatives.

### Overall Comparison

We now compare the performance of our selected solution 2Tp against the competitive approaches HDT-FoQ [Martínez-Prieto et al., 2012] and TripleBit [Yuan et al., 2013]. We use the C++ libraries provided by the corresponding authors: https://github.com/rdht/hdt-cpp, and https://github.com/nitingupta910/TripleBit, respectively. Table III 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× (TripleBit does not support the query pattern SPO). Concerning the space, we see that the 2T 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× and, depending on the pattern examined, we report peaks of 26×, 49×, 81×, or even 2057×.

### REFERENCES

