Compressed Indexes for Fast Search of Semantic Data*

Raffaele Perego  
ISTI-CNR  
Pisa, Italy

Giulio Ermanno Pibiri  
ISTI-CNR  
Pisa, Italy

Rossano Venturini  
University of Pisa and ISTI-CNR  
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.

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 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×.

1 THE PERMUTED TRIE INDEX
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 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 ?O 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. 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.

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. The described index layout represents the triples three times in different (cyclic) permutations in order to optimally solve all triple selection patterns. However, the data structure does not take advantage of the fact that the same set of triples is represented multiple times and, consequently, the index has an abundance of redundant information. 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 2 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 \( p_i \) in the set, we determine if the object \( o \) is a child of \( p_i \) with a search operation: if it is, then \( (s, p_i, o) \) is a triple to return. The correctness of the algorithm is immediate and we argue that its efficiency is due to the following facts.

The small number of predicates as children of a given subject implies that the algorithm will perform few iterations: while these children are few per se (for example, at most 52 for the DBpedia dataset), the iterations will be on average far less. For a given \( (subject, predicate) \) pair, we have a very limited number of children to be searched for \( o \), thus making the search operation run faster by short scans rather than via binary search.

In the light of the just described 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 ?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 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).

---

1 Extended abstract of the paper [2].
We perform our experimental analysis on large and publicly available standard datasets, whose statistics are summarized in Table 2.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Triples</th>
<th>Subjects (S)</th>
<th>Predicates (P)</th>
<th>Objects (O)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBLP</td>
<td>88,150,324</td>
<td>5,125,936</td>
<td>27</td>
<td>36,413,780</td>
</tr>
<tr>
<td>Geonames</td>
<td>123,020,821</td>
<td>8,345,459</td>
<td>26</td>
<td>42,728,317</td>
</tr>
<tr>
<td>DBpedia</td>
<td>351,592,624</td>
<td>27,318,781</td>
<td>1480</td>
<td>115,872,941</td>
</tr>
<tr>
<td>Freebase</td>
<td>2,067,068,154</td>
<td>102,001,451</td>
<td>770,415</td>
<td>438,832,462</td>
</tr>
</tbody>
</table>

Table 2: Datasets statistics.

All the experiments are performed on a server machine with 4 Intel i7-7700 cores (@3.6 GHz), 64 GB of RAM DDR3 (@0.133 GHz) and running Linux 4.4.0, 64 bits. The C++ implementation of our indexes is freely available at https://github.com/jermp/rdf_indexes. We compiled the code with gcc 7.3.0 using the highest optimization setting, i.e., with compilation flags -O3 and -march=native.

To measure the query processing speed, we used a set of 5000 triples drawn at random from the datasets and set 0, 1 or 2 wildcard symbols. In all tables, percentages and speed up factors are taken with respect to the values highlighted in bold font.

The permuted trie index. By looking at the results reported in Table 1, 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 ?P?. 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.

Table 3: 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.

**REFERENCES**

