Compressed Indexes for Fast Search of Semantic Data

Raffaele Perego  
ISTI-CNR  
Pisa, Italy

Giulio Ermanno Pibiri  
ISTI-CNR  
Pisa, Italy

Rossano Venturini  
The University of Pisa  
Pisa, Italy

The 10-th Italian Information Retrieval Workshop (IIR 2019)  
17/09/2019
“RDF is a standard model for data interchange on the Web.”
Source: https://www.w3.org/RDF

Statements are encoded with **triples**: Subject (S) - Predicate (P) - Object (O)
“RDF is a standard model for data interchange on the Web.”
Source: https://www.w3.org/RDF

Statements are encoded with **triples**: 
Subject (S) - Predicate (P) - Object (O)

“Bob Smith knows John Doe.”

<http://example.name#BobSmith12> <http://xmlns.com/foaf/0.1/knows> <http://example.name#JohnDoe34>
The problem

Huge datasets: **billions** of triples.

Storage space is an issue: 
**compression is mandatory.**

How to support triple selection patterns (with wildcards) **efficiently**?
Huge datasets: billions of triples.

Storage space is an issue: compression is mandatory.

How to support triple selection patterns (with wildcards) efficiently?

<Bob Smith> <knows> <???>

<???> <???> John Doe

<Bob Smith> <???> <Sara Parker>
The problem

Huge datasets: **billions** of triples.

Storage space is an issue: **compression is mandatory**.

How to support triple selection patterns (with wildcards) **efficiently**?

```
<Bob Smith> <knows> <????>
<????> <????> John Doe
<Bob Smith> <????> <Sara Parker>
```

```
1 wildcard: SP?
2 wildcards: S??
1 wildcard: S?O
2 wildcards: ?P?
1 wildcard: ?PO
2 wildcards: ??O
3 wildcards: ???
0 wildcard: SPO
```
State-of-the-art solutions

Too costly in terms of \textit{space}.

\begin{itemize}
  \item Materialize \textbf{all} possible S-P-O permutations (6 separate indexes).
  \item \textbf{Do not} use sophisticated compression techniques.
  \item Expensive additional indexes to support retrieval.
\end{itemize}
Map URI strings to integers to reduce space requirements: we deal with datasets of integer triples.

Selection patterns

S P O
S P ?
S ??
?? ?

? P O
? P ?
S ? O
?? O
Map URI strings to integers to reduce space requirements: we deal with datasets of integer triples.

Selection patterns

S P O
S P ?
S ??
??

? P O
? P ?

S ? O
?? O

S-P-O order
The Permuted Trie Index: preliminaries

Map URI strings to integers to reduce space requirements:
we deal with datasets of integer triples.

Selection patterns

- S P O
- S P ?
- S ??
- ?? ?

→ S-P-O order

- ? P O
- ? P ?

→ P-O-S order
Map URI strings to integers to reduce space requirements: we deal with datasets of integer triples.

**Selection patterns**

- \( S \text{ P O} \) → \( S-P-O \) order
- \( ? \text{ P O} \) → \( P-O-S \) order
- \( S ? ? \text{ O} \) → \( O-S-P \) order
The Permuted Trie Index: preliminaries

Map URI strings to integers to reduce space requirements: we deal with datasets of integer triples.

Selection patterns

```
S P O
S P ?
S ??
???

? P O
? P ?

S ? O
?? O

S-P-O order
P-O-S order
O-S-P order
```

Store an integer trie data structure for each permutation.
The Permuted Trie Index: organisation

```
levels[0]   0  2  4  6  7  8
levels[1]   0  1  0  2  0  1  2  2
levels[2]   2  3  0  4  0  1  2  0  1  2  4
```

Diagram showing the structure of the permuted trie index.
- **Common prefixes** are encoded once.
- Two integer **sequences** per level (nodes and pointers).
- Symmetrically support all selection patterns with 1 and 2 wildcards.
- **Cache-friendly** memory layout.
The Permuted Trie Index: organisation

- **Common prefixes** are encoded once.
- Two integer **sequences** per level (nodes and pointers).
- Symmetrically support **all** selection patterns with 1 and 2 wildcards.
- **Cache-friendly** memory layout.
The Permuted Trie Index: organisation

- Common prefixes are encoded once.
- Two integer sequences per level (nodes and pointers).
- Symmetrically support all selection patterns with 1 and 2 wildcards.
- Cache-friendly memory layout.

Allows effective compression

Fast retrieval
The Permuted Trie Index: organisation

- Common prefixes are encoded once.
- Two integer sequences per level (nodes and pointers).
- Symmetrically support all selection patterns with 1 and 2 wildcards.
- Cache-friendly memory layout.
The Permuted Trie Index: organisation

- **Common prefixes** are encoded once.

- Two integer **sequences** per level (nodes and pointers).

- Symmetrically support all selection patterns with 1 and 2 wildcards.

- **Cache-friendly** memory layout.

**Allows effective compression**

**Fast retrieval**
The Permuted Trie Index: organisation

- Common prefixes are encoded once.
- Two integer sequences per level (nodes and pointers).
- Symmetrically support all selection patterns with 1 and 2 wildcards.
- Cache-friendly memory layout.

Allows effective compression

Fast retrieval
The Permuted Trie Index: refinements

1. Cross Compression
2. Permutation Elimination
Fact: the **same** triple appears three times, but in **different** permutations.
Cross Compression

Fact: the **same** triple appears three times, but in **different** permutations.

We can represent the subjects in trie 1 by using the subjects in trie 2.
Fact: the **same** triple appears three times, but in **different** permutations.

We can represent the subjects in trie 1 by using the subjects in trie 2.
Fact: the **same** triple appears three times, but in **different** permutations.

We can represent the subjects in trie 1 by using the subjects in trie 2.

Represent $S_j$ as its position $p$. 
Fact: the same triple appears three times, but in different permutations.

We can represent the subjects in trie 1 by using the subjects in trie 2.

Represent $S_j$ as its position $p$.

### Why?

<table>
<thead>
<tr>
<th>Trie</th>
<th>Level</th>
<th>Average</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPO</td>
<td>1</td>
<td>5.54</td>
<td>52</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2.32</td>
<td>8489</td>
</tr>
<tr>
<td>POS</td>
<td>1</td>
<td>91,578.32</td>
<td>21,219,244</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2.59</td>
<td>10,141,311</td>
</tr>
<tr>
<td>OSP</td>
<td>1</td>
<td>2.70</td>
<td>10,141,327</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.13</td>
<td>10</td>
</tr>
</tbody>
</table>

Number of children in Dbpedia.
Fact: predicates are few, thus S?O returns only few matches.
Fact: predicates are few, thus S?O returns only few matches.

We can **pattern match S?O on the SPO trie**, instead of the OSP trie.

Given a \((s,o)\) pair: for each child \(p_i\) of \(s\), check if \(o\) is a child of \(p_i\). If so, then \((s,p_i,o)\) is a match.
Fact: predicates are few, thus S?O returns only few matches.

We can pattern match S?O on the SPO trie, instead of the OSP trie.

Given a (s,o) pair: for each child $p_i$ of s, check is o is a child of $p_i$. If so, then (s,$p_i$,$o$) is a match.

<table>
<thead>
<tr>
<th>Trie</th>
<th>Level</th>
<th>Average</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPO</td>
<td>1</td>
<td>5.54</td>
<td>52</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2.32</td>
<td>8489</td>
</tr>
<tr>
<td>POS</td>
<td>1</td>
<td>91,578.32</td>
<td>21,219,244</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2.59</td>
<td>10,141,311</td>
</tr>
<tr>
<td>OSP</td>
<td>1</td>
<td>2.70</td>
<td>10,141,327</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.13</td>
<td>10</td>
</tr>
</tbody>
</table>

Number of children in Dbpedia.

Less than 6 checks are needed on average!
Permutation Elimination

SPO trie

S P O
S P ?
S ??
S ? O
???

+ OR

OR
Permutation Elimination

SPO trie

OPS trie

Object-based retrieval

OR

+
Permutation Elimination

SPO trie

OPS trie

POS trie

Object-based retrieval

Predicate-based retrieval

OR
Permutation Elimination

We can eliminate a permutation, thus saving 1/3 of the space of the index.
Experiments: setting

Datasets

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Triples</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBLP</td>
<td>88,150,324</td>
</tr>
<tr>
<td>Geonames</td>
<td>123,020,821</td>
</tr>
<tr>
<td>DBpedia</td>
<td>351,592,624</td>
</tr>
<tr>
<td>Freebase</td>
<td>2,067,068,154</td>
</tr>
</tbody>
</table>

Machine

i7-7700 CPU (@3.6 GHz), 64 GB of RAM DDR3 (@2.133 GHz)
Linux 4.4.0, 64 bits

Compiler

gcc 7.2.0 (with all optimizations)
Indexes for RDF data

This is the C++ library used for the experiments in the paper *Compressed Indexes for Fast Search of Semantic Data* [1], by Raffaele Perego, Giulio Ermanno Pibiri and Rossano Venturini.

This guide is meant to provide a brief overview of the library and to illustrate its functionalities through some examples.

Table of contents

1. Compiling the code
2. Input data format
3. Preparing the data for indexing
4. Building an index
5. Querying an index
6. Statistics
7. Testing
8. Extending the software
9. Authors
10. References
## Experiments: our solutions

<table>
<thead>
<tr>
<th></th>
<th>DBLP</th>
<th>Geonames</th>
<th>DBpedia</th>
<th>Freebase</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>bits/triple</td>
<td>bits/triple</td>
<td>bits/triple</td>
<td>bits/triple</td>
</tr>
<tr>
<td>3T</td>
<td>75.24 (+31%)</td>
<td>71.59 (+32%)</td>
<td>80.64 (+33%)</td>
<td>74.20 (+30%)</td>
</tr>
<tr>
<td>CC</td>
<td>63.54 (+18%)</td>
<td>67.04 (+27%)</td>
<td>66.91 (+19%)</td>
<td>70.46 (+26%)</td>
</tr>
<tr>
<td>2To</td>
<td>56.46 (+8%)</td>
<td>53.23 (+8%)</td>
<td>57.51 (+6%)</td>
<td>55.72 (+6%)</td>
</tr>
<tr>
<td>2Tp</td>
<td>51.99</td>
<td>48.98</td>
<td>54.14</td>
<td>52.17</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>ns/triple</th>
<th>ns/triple</th>
<th>ns/triple</th>
<th>ns/triple</th>
</tr>
</thead>
<tbody>
<tr>
<td>3T,CC</td>
<td>2490 (5.6×)</td>
<td>3767 (7.7×)</td>
<td>1833 (2.6×)</td>
<td>6547 (1.8×)</td>
</tr>
<tr>
<td>2To,2Tp</td>
<td>445</td>
<td>490</td>
<td>692</td>
<td>3736</td>
</tr>
<tr>
<td>3T,2To,2Tp</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>CC</td>
<td>12 (2.4×)</td>
<td>15 (3.0×)</td>
<td>16 (3.2×)</td>
<td>14 (2.8×)</td>
</tr>
<tr>
<td>3T,CC</td>
<td>12 (2.4×)</td>
<td>12 (2.4×)</td>
<td>12 (2.4×)</td>
<td>10 (2.0×)</td>
</tr>
<tr>
<td>2To</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>2Tp</td>
<td>5 (1.0×)</td>
<td>5 (1.0×)</td>
<td>6 (1.2×)</td>
<td>10 (2.0×)</td>
</tr>
<tr>
<td>3T,2Tp</td>
<td>9 (2.3×)</td>
<td>8 (4.5×)</td>
<td>6 (5.0×)</td>
<td>6 (4.8×)</td>
</tr>
<tr>
<td>CC</td>
<td>21 (9.0×)</td>
<td>36 (17.2×)</td>
<td>22 (3.7×)</td>
<td>18 (3.0×)</td>
</tr>
</tbody>
</table>

Overall, 2Tp offers the best space/time tradeoff.
Our selected trade-off configuration substantially outperforms the tested competitors in both space and time.

<table>
<thead>
<tr>
<th>Index</th>
<th>DBLP (bits/triple)</th>
<th>Geonames (bits/triple)</th>
<th>DBpedia (bits/triple)</th>
<th>Freebase (bits/triple)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2Tp</td>
<td>51.99</td>
<td>48.98</td>
<td>54.14</td>
<td>52.17</td>
</tr>
<tr>
<td>HDT-FoQ</td>
<td>76.89 (+32%)</td>
<td>88.73 (+45%)</td>
<td>76.66 (+29%)</td>
<td>83.11 (+37%)</td>
</tr>
<tr>
<td>TripleBit</td>
<td>125.10 (+58%)</td>
<td>120.03 (+59%)</td>
<td>130.07 (+58%)</td>
<td>—</td>
</tr>
<tr>
<td>ns/triple</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2Tp</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>HDT-FoQ</td>
<td>12 (2.4×)</td>
<td>13 (2.6×)</td>
<td>14 (2.8×)</td>
<td>13 (2.6×)</td>
</tr>
<tr>
<td>TripleBit</td>
<td>15 (3.0×)</td>
<td>13 (2.6×)</td>
<td>14 (2.8×)</td>
<td>—</td>
</tr>
<tr>
<td>S?O</td>
<td>445</td>
<td>490</td>
<td>692</td>
<td>3736</td>
</tr>
<tr>
<td>HDT-FoQ</td>
<td>1789 (4.0×)</td>
<td>2097 (4.3×)</td>
<td>3010 (4.3×)</td>
<td>0.7×10⁷ (2057×)</td>
</tr>
<tr>
<td>TripleBit</td>
<td>11872 (26.7×)</td>
<td>13008 (26.5×)</td>
<td>18023 (26.0×)</td>
<td>—</td>
</tr>
<tr>
<td>SP?</td>
<td>197</td>
<td>347</td>
<td>11</td>
<td>3</td>
</tr>
<tr>
<td>HDT-FoQ</td>
<td>640 (3.2×)</td>
<td>897 (2.6×)</td>
<td>30 (2.7×)</td>
<td>9 (3.0×)</td>
</tr>
<tr>
<td>TripleBit</td>
<td>1222 (6.2×)</td>
<td>927 (2.7×)</td>
<td>42 (3.8×)</td>
<td>—</td>
</tr>
<tr>
<td>S??</td>
<td>28</td>
<td>40</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>HDT-FoQ</td>
<td>110 (3.9×)</td>
<td>154 (3.9×)</td>
<td>29 (2.9×)</td>
<td>9 (3.0×)</td>
</tr>
<tr>
<td>TripleBit</td>
<td>2275 (81.2×)</td>
<td>3261 (81.5×)</td>
<td>490 (49.0×)</td>
<td>—</td>
</tr>
<tr>
<td>?P?</td>
<td>9</td>
<td>8</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>HDT-FoQ</td>
<td>108 (12.0×)</td>
<td>173 (21.6×)</td>
<td>32 (5.3×)</td>
<td>41 (6.8×)</td>
</tr>
<tr>
<td>TripleBit</td>
<td>28 (3.1×)</td>
<td>28 (3.5×)</td>
<td>40 (6.7×)</td>
<td>—</td>
</tr>
<tr>
<td>?O</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>HDT-FoQ</td>
<td>17 (3.4×)</td>
<td>17 (3.4×)</td>
<td>18 (3.0×)</td>
<td>18 (1.8×)</td>
</tr>
<tr>
<td>TripleBit</td>
<td>24 (4.8×)</td>
<td>60 (12.0×)</td>
<td>24 (4.0×)</td>
<td>—</td>
</tr>
</tbody>
</table>
Conclusions

The *triple indexing problem with pattern matching* can be solved efficiently in both time and space regards.

Our solution — the **permuted trie index** — achieves substantial performance improvement against the best previous solutions.

**Cross-compression**  
**Permutation-elimination**

Paper available at  
C++ code available at  
https://github.com/jermp/rdf_indexes
Thanks for your attention, time, patience!

Any questions?