Compression of RDF Dictionaries

Miguel A. Martínez-Prieto
Dept. of Computer Science
Univ. of Valladolid, Spain
Dept. of Computer Science
Univ. of Chile, Chile
migumar2@infor.uva.es

Javier D. Fernández
Dept. of Computer Science
Univ. of Valladolid, Spain
Dept. of Computer Science
Univ. of Chile, Chile
jfergar@infor.uva.es

Rodrigo Cánovas
NICTA Victoria Research Lab.
Dept. of Computer Science
and Software Engineering
Univ. of Melbourne, Australia
Dept. of Computer Science
Univ. of Chile, Chile
rcanovas@student.unimelb.edu.au

ABSTRACT

The use of dictionaries is a common practice among those applications performing on huge RDF datasets. It allows long terms occurring in the RDF triples to be replaced by short IDs which reference them. This decision greatly compacta the dataset and thus mitigates its scalability issues. However, the dictionary size is not negligible and the techniques used for its representation also suffer from scalability limitations. This paper focuses on this scenario by adapting compression techniques for string dictionaries to the case of RDF. We propose a novel technique: $D_{\text{comp}}$, which can be tuned to represent the dictionary in compressed space (22–64%) and to perform in a few microseconds (1–50µs).

Keywords

RDF Dictionaries, Scalability, Compression, SPARQL

1. INTRODUCTION

Nowadays, the so-called Web of Data materializes the basic principles of the Semantic Web [8]. It interconnects datasets from diverse fields of knowledge within a cloud of data-to-data hyperlinks which enables a ubiquitous and seamless data integration to the lowest level of granularity. As the Web of Data grows in popularity, the number (and scale) of semantic applications in use increases, more data are linked together and larger datasets are increasingly obtained. In this context, performance and scalability arise as major issues and their resolution is closely related to the efficient storage and retrieval of the data.

The World Wide Web Consortium (W3C) recommends the use of the RDF[2] (Resource Description Framework) data model for conceptual description and SPARQL[3] as the querying language. RDF provides a graph-based model for structuring and linking data which describes facts of the world [9]. It is based on atomic triples comprising a subject $s$, (the resource being described), a predicate $p$, (the property), and an object $o$ (the property value). A set of RDF triples makes up an RDF graph in which the knowledge is represented through the different terms stored in nodes and edges. This term collection (called vocabulary) comprises elements drawn from three disjoint classes: Uniform Resource Identifiers (URIs), blank nodes (bnodes), and literals.

The vocabulary of terms is commonly indexed through a bijective function (called dictionary) mapping the strings representing the terms and integer IDs. Thus, a dictionary gives two complementary operations: (i) string-to-ID returns the ID of a given string, and (ii) ID-to-string retrieves the string corresponding to a given ID. Both operations are used by SPARQL engine during the query process.

The use of dictionaries is a simple but effective decision for managing RDF, because all triples in the dataset can be rewritten by replacing the terms with their corresponding ID. Thus, the original dataset is now modeled through the dictionary and the resultant ID-triples representation. It allows high compression ratios to be achieved [22].

Despite of the undeniable contribution of dictionaries for improving scalability, their use is also compromised in the current scenario. As shown in Section 5.1, the space required by the dictionaries is even larger than that used for the resulting ID-triples representations, so managing their scalability is an issue by its own. Whereas much research work has been developed for compression and/or indexing of the ID-triples representations (see section 3), specific compressed dictionary representations are not covered in the previous literature to the best of our knowledge. Our main contributions in this scenario are:

- An introduction to the problem of effective representations of RDF dictionaries together with an empirical study characterizing their main features regarding an RDF dataset.
- A practical deployment of how compressed string dictionaries are used for representing RDF vocabularies.
- A configurable technique (called $D_{\text{comp}}$) which achieves highly-compressed RDF dictionaries and very efficient performance.

The paper is structured as follows. Section 2 shows different approaches for compressed string dictionaries, and Section 3 describes RDF dictionaries and reviews their state-of-the-art. Our approach for compressed RDF dictionaries...
(called \(D_{\text{comp}}\)) is explained in Section 4. Then, Section 5 shows an empirical study of RDF dictionaries and analyzes the \(D_{\text{comp}}\) performance for five real-world datasets. Finally, Section 6 gives conclusions on the current results and devises future lines of work in this field.

2. COMPRESSED STRING DICTIONARIES

String dictionaries are the natural precedent of RDF dictionaries. A string dictionary \(D = \{s_1, s_2, \ldots, s_n\}\) contains all different strings (vocabulary) representing the terms used in a dataset. Its basic handling is reduced to the efficient answering of two queries: locate\((s_i)\) which maps the string \(s_i\) into its ID in \(D\) (string-to-ID), and extract\((i)\) which returns the string \(s_i\) identified as \(i\) in \(D\) (ID-to-string).

Classical approaches for string dictionaries, like hashing \cite{10}, use much space. It dissuades applications handling large vocabularies (for instance, the Web of Data) to use it, because of the limited size of available memory. The use of B-tree \cite{6} based solutions is the alternative considering their optimization for large scale disk representations. However, their efficiency is compromised by the I/O costs derived from disk transfers. In this scenario, compression arises as the natural solution for increasing the amount of data which can be efficiently managed in memory.

Bender et al. \cite{7} propose a cache-oblivious string B-tree performing Front-Coding \cite{25} compression in the leaves. It is improved by the compressed permuterm \cite{15} which gives efficient support for locate and extract and also resolves some substring-based operations in a compressed space. A more recent work, by Brisaboa et al. \cite{10}, revisits the problem from an eminently practical perspective. It proposes compressed variants of well-known solutions and introduces some novel ones. These are studied for emergent applications (for instance, they test a dictionary of URIs) and their results guarantee their interest in the RDF scenario.

We consider three types of techniques to give a more complete coverage of the current scenario: (\S2.1) Hashing is a classical solution for representing any kind of dictionary; (\S2.2) Front-Coding takes advantage of the existence of long common prefixes to obtain compressed dictionaries; and (\S2.3) Self-indexes are general compressed indexes suitable to be adapted for representing string dictionaries. All these techniques are shown by following their original description \cite{10}. They process the dictionary as a text \(T_{\text{dict}}\) which concatenates all strings ended by a special $ symbol (it is, in practice, the ASCII zero code).

2.1 Hashing

Hashing \cite{12} is a natural choice for representing key-value structures like that required for a dictionary. It excels for locate because the hash function is a natural way to transform a string into an ID, yet it has no a primitive mechanism to answer extract. Besides, hashing does not achieve compression by itself. On the one hand, it needs space for storing all \(m\) strings in the dictionary. On the other hand, extra storage space is required for representing the hash table \(H[1, n]\). The value \(m/n\) (\(m < n\)) is the load factor and it influences the space usage and the time performance.

We consider the technique named as HashB-dh in \cite{10} (we rename it as Hash) because it yields the best space/time tradeoffs from among the hashing-based ones. It achieves compression through two basic decisions:

- It stores the hash table in a compact form by removing all the empty cells: \(H'[1, m]\). A bitmap structure \(B[1, n]\) is now required: \(B[i] = 1\) if \(H'[i]\) is a non-empty cell and \(B[i] = 0\) if \(H'[i]\) is empty.
- It compresses \(T_{\text{dict}}\) with Huffman \cite{19} and performs the hash function over the compressed strings.

Besides, this technique integrates a primitive resolution for extract. It sorts \(T_{\text{dict}}\) to store the strings in the same order used in \(H'\). Thus, extract is answered by directly accessing the corresponding cell in the hash table.

2.2 Front-Coding

Front-Coding \cite{25} is a traditional technique for compressing lexicographically sorted dictionaries. It takes advantage of consecutive strings which are likely to share a common prefix and they can be differentially encoded. The dictionary is partitioned into buckets of \(b\) strings to allow efficient querying (a space/time tradeoff is yielded through \(b\) values).

A Plain Front-Coding (PFC) is firstly defined in \cite{10}. It uses a byte-oriented encoding for compressing the differential representation of each string. This decision allows performance to be improved because all queries can be completed by exclusively running fast byte-wise operations. In turn, the Hu-Tucker Front-Coding (HTFC) focuses on spatial effectiveness. It compresses the byte-stream with Hu-Tucker \cite{20} and performs on the compressed representation. This decision allows significative size reductions to be attained at the price of slightly increasing querying times.

Both techniques perform on a lexicographic \(T_{\text{dict}}\) ordering. This means they are especially suitable for vocabularies containing strings with long common prefixes.

2.3 Self-Indexing

Self-indexes \cite{21} take advantage of the compressibility of a text to represent it in a structure that uses space closer to the compressed text, providing search functionality and containing enough information to reproduce any text substring. Thus, a self-index can replace the text. The FM-Index \cite{14} is a self-index modeling the text on the Burrows-Wheeler Transform (BWT) \cite{11}. This transformation is the core of the well-known compressor bzip2, so it gives a notion of the FM-Index effectiveness for compressing general texts.

An FM-Index (FM1) based approach is proposed in \cite{10} for compressing string dictionaries. It also performs on a lexicographic \(T_{\text{dict}}\) ordering and achieves effective compressed representations for all studied scenarios at the price of a less competitive performance for locate and extract.

3. RDF DICTIONARIES

An RDF dictionary organizes all different terms used in a dataset. They come from three disjoint classes:

- **URIs** (\(U\)) identify resources in the WWW. Thus, these are the identifiers used for data integration in the Web of Data. Many terms in the URI set share long prefixes.
- **Bnodes** (\(B\)) name anonymous nodes in the RDF graph and usually serve as parent nodes to a grouping of data. Naming of bnodes can matter in some treatments. Thus, canonical representations of RDF are due to the structure of bnodes which are in general
tricky to achieve. For our purpose, we consider the
bnodes naming convention of N3.

- **Literals** (L) can be considered as “end nodes” in RDF
graphs because they play the object role. Although
literals can be tagged with an optional language or
datatype, no general features can be assumed about
their content. It is strongly related to the knowledge
represented in the dataset.

The RDF model does not allow the subject to be a literal
and the predicate must be an URI. All types can play the
object role. Thus, an RDF triple \((s, p, o) \in (U \cup B) \times (U) \times
(U \cup B)\).

**State-of-the-art.**

The Web of Data popularity is basis for the development
of RDF management systems (referred as RDF stores) that
provide efficient storage and lookup infrastructure. As
explained in Section 1, dictionaries are used for compression
purposes, but their specific representation is also an issue for
querying: SPARQL engines make use of dictionary indexes,
in conjunction with evaluation and histogram indexes, for
physical optimization [17].

SPARQL resolution and RDF dictionaries are clearly re-
lated. SPARQL considers *triple patterns* (i.e. RDF triples
\((s, p, o)\) in which \(s, p, o\) may be a variable) as atomic
queries for building more complex ones. Thus, the engine
firstly *locates* the IDs associated to the terms given in the
triple patterns. The transformed query is then performed
and the resulting ID values bound to the variables given
in this query. The final result is obtained by *extracting*
the terms associated to these resulting IDs. This basic pro-
cess implies that *locate* and *extract* are unevenly used.
Whereas *extract* is used many times as results are returned
for each variable in the query, the use of *locate* is limited to
the number of terms bounded in the query. Thus, *extract*
tends to be overused in comparison to *locate* and should,
in general, be optimized.

Many real-world RDF stores, such as the C-store based
one [5], Hexastore [24], or RDF-3X [22] among others, create
and maintain dictionary indexes. However, any store gives
optimized dictionary solutions. All of them use two inde-
dependent structures, giving ineffective solutions which dou-
ble the space required for representing the dictionary. For
*locate*, B*-tree disk-oriented based solutions are used for
term to ID mappings. In some cases, Front-Coding com-
pression is performed on the leaves. For *extract*, structures
supporting constant-time direct access (such as arrays or
memory-mapped files) are used for ID to term mappings.

Dictionaries are also a core element for RDF exchang-
ing. The W3C Member Submission HDT [4] is an RDF data-
centric format which reduces verbosity in favor of machine-
understandability and data management. It represents a
dataset through a *Dictionary* which arranges all strings in
the dataset, and a *Triples* component which models the ID-
based representation of the RDF graph topology. The most-
native HDT representation (without any kind of compres-
sion) reduce the dataset size up to fifteen times [13].

Some other specific applications within the Web of Data
perform by using RDF dictionaries. We emphasize its use in
reasoning applications. In this scenario, Hogan [18] claims
that a dictionary of URIs requires a prohibitive amount of
memory to be stored and its compression would help to in-
crease the in-memory capacity. It is very relevant for our
objectives to consider that in-memory representations sup-
port higher degree of reasoning.

4. **OUR APPROACH: D\(_{\text{COMP}}\)**

An RDF dictionary technique must be optimized from
two correlated perspectives: i) the space used for its rep-
resentation, and ii) the time required for answering *locate*
and *extract*. On the one hand, the spatial perspective is re-
lated to the URI, bnode and literal vocabularies; a dictionary
technique which detects and compresses specific vocabulary
regularities allows spatial requirements to be optimized. On
the other hand, query resolution depends on the efficient
translation of the terms given in the triple patterns (*locate*),
and the returned query solutions (*extract*). Both operations are performed by attending to the role played by
*terms* and *variables* in the triple patterns.

Our approach \(D_{\text{COMP}}\) considers a specific organization com-
bing these two perspectives. A role-based partitioning is
firstly considered and all terms in the dictionary are organ-
ized according to the role they play in the dataset:

- **Common subjects and objects** (S0) organizes all
terms which play subject and object roles in the dataset.
They are mapped to the range \([1, |S0]|\).

- **Subjects** (S) organizes all subjects which do not play
an object role. They are mapped to \([|S0|+1, |S0|+|S|]\).

- **Objects** (O) organizes all objects which do not play a
subject role. They are mapped to \([|S0|+|S|, |S0|+|0|]\).

- **Predicates** (P) maps all predicates to \([1, P]\).

Note that a given ID can belong to different sets, but the
disambiguation of the correct set is trivial when we know if
the ID to search is a subject, a predicate or an object.

The partition S0 allows terms playing subject and object
roles to be represented once. This decision achieves spatial
 savings as shown in Section 5.1: up to 60% of the strings
may be in S0. This role-based organization allows \(D_{\text{COMP}}\)
to perform on three ID-range mapping terms in \([1, |S0|+|S|]\)
(subjects), \([1, |S0|+|O|]\) (objects), and \([1, |P|]\) (predicates).

An internal subdivision is then performed by attending to
the classes \((U, B, L)\) that each role-partition can store. It
allows the technique which best adjusts each class in accord-
dance to its features and its application requirements to be
chosen. That is, a specific dictionary is used to represent
the terms for each class within each partition. This implies
that each dictionary handles its specific mapping, so each
term is locally identified within it.

Figure 1 illustrates the resulting organization. As can be
seen, the partitions \(S=0\) and \(S\) are split into URIs and bnodes.
In turn, \(O\) contains URIs and bnodes, but also an area for
literals in which specific representations for *general* literals,
*lang* literals (tagged with a specific language), and *datatype*
literals (tagged with its specific datatype) are maintained.
These last two groups are subdivided to classify all different
languages and datatypes used in the dataset. This hierar-
chy delimits the ranges for a given language or datatype,
allowing string tags to be removed in the final literal repre-
sentation. The partition \(P\) only contains URIs.

This organization requires a very small mapping structure
(referred to as *ptrs*) for integrating all partitions in a single
A simple example in considered to illustrate this process. Let us suppose the term $t_i = \text{“Santiago, Chile”};$ it is drawn from the class of literals and plays, necessarily, the object role. First, the processor narrows the search to the 0 partition, and then determines the dictionary of literals which must be queried. As shown in Figure 1, information for this dictionary is stored in the Cell $\text{ptrs}[7].$ In this case, the term represents a general literal, so this is the dictionary finally queried. The operation $\text{locate}(t_i)$ returns the local ID: $id_x,$ for $t_i.$ It is then transformed into its corresponding global ID as $x = \text{ptrs}[7] + id_x,$ and this is the value finally returned to the query processor.

**Extract.** This process retrieves the term associated to a given ID $y$ returned in the result set of a query. Note that $y$ is a global ID, so it can be directly used to determine the corresponding dictionary. It is easily implemented through a binary search comparing $y$ against the values in $\text{ptrs}.$ The ID is represented in the dictionary $D_k$ pointed out by the cell $\text{ptrs}[k],$ $\text{ptrs}[k] \leq y < \text{ptrs}[k+1].$ The ID $j$ is now transformed into its corresponding local ID as $id_y = y - \text{ptrs}[k],$ and the primitive $\text{extract}(id_y)$ is performed to retrieve the desired term. It is bound to its corresponding variable in $Q$ and returned in the final result set. It is worth noting that determining the appropriate subdictionary for literals requires another binary search on the second level of $\text{ptrs}.$ Let us suppose that the result set of a query contains the ID $y = 216.$ The extraction process firstly performs a binary search which determines that $y$ is represented in the class of URIs within the $S$ partition ($\text{ptrs}[2]).$ This happens when $\text{ptrs}[2] \leq y < \text{ptrs}[3].$ The global ID 216 is then transformed into its local counterpart $(id_y = \text{ptrs}[2] - 216),$ and the primitive $\text{extract}(id_y)$ is performed, returning the term $<$http://dbpedia.org/resource/Santiago,_Chile$>.$

**5. EXPERIMENTAL SETUP**

This section studies the problem of compression of RDF dictionaries on a real-world setup. To do this, we choose five real-world datasets to achieve a heterogeneous setup containing data from different application domains. The amount of triples in each dataset is also considered in our choice. Three different datasets are extracted from the Billion Triples Challenge 2010\footnote{http://km.aifb.kit.edu/projects/btc-2010/}; in particular, geonames gathers geographic concepts, dbtune holds music data and uniprot contains biological information mainly focused on proteins; wikipedia\footnote{http://labs.syscomone.at} stands for the English Wikipedia links between pages transformed to RDF, and dbpedia\footnote{http://wiki.dbpedia.org/downloads36} is a community effort with the aim of making this type of information semantically available on the Web.

We firstly characterize the RDF dictionaries extracted from these datasets (§5.1). We focus on the impact of the dictionary size within a dataset and also study their more relevant statistics to the current problem. Then (§5.2), we test compressed string dictionaries in the RDF scenario and extract conclusions for $D_{\text{comp}}.$ It is studied in (§5.3) through two functional configurations: $D_{\text{comp}}^C$ is focused on compression effectiveness and $D_{\text{comp}}^Q$ is optimized for querying.

All querying tests are performed on a computer using an Intel Core2 Duo processor at 3.16 GHz, with 8 GB of main
memory and 6 MB of cache, running Linux kernel 2.6.24-28. User times are reported for all experiments. Prototypes are developed in C++ using structures from libcds[1]. Plain (referred to as RG [16]) and compressed (referred to as RRR [23]) bitmaps are tested in our experiments; they can be parameterized with a sampling value. All sources are compiled on g++ 4.2.4 with options -O9 and -m64.

### 5.1 Characterizing RDF Dictionaries

The dictionary size is the issue addressed in this paper under the consideration that it is a significant fraction of the dataset size. This assumption can be confirmed by transforming the original dataset into an equivalent raw representation of dictionary and ID-triples: the dictionary concatenates all terms ended by a separator symbol and the ID-triples representation replaces the terms by their corresponding ID and performs a naive encoding which uses \( \log(n) \) bits/element (n is respectively the number of total subjects, predicates or objects respectively). We assume that dictionary and ID-triples representations use \( c_d \) and \( c_t \) bytes respectively, so the dataset size is \( C = c_d + c_t \) bytes.

Table 1 (left) shows this comparison. Columns \( C \) and \( C' \) respectively contain the size (in GB) of the original N3 dataset and that obtained through the dictionary-based representation. As can be seen, the use of dictionaries allows for large compression (e.g. for wikipedia, the dictionary-based representation is 11.80 times smaller than the original). The next two columns measure the impact of the dictionary size in this representation (values in parentheses correspond to \( c_d/C \) and \( c_t/C \) respectively). As can be seen, \( c_d \) is always larger than \( c_t \) (up to \( \approx 3 \) times for geonames or dbpedia). Thus, the dictionary always takes more space than the most-naive ID-based representation. This fact supports the need for effective dictionary representations that can be used in conjunction with more-advanced techniques for ID-triples.

Table 1 (right) describes the studied datasets. The columns triples and elements respectively show the amount of triples in the dataset and the number of terms contained in the vocabulary. As can be seen, this vocabulary grows with the dataset size, but the proportion depends on the design and purpose of the dataset. The next columns contain the distribution of terms playing different roles in the dataset. Again, their proportion depends on the dataset features but two significative conclusions can be extracted: i) the SO partition allows terms playing roles of subject and object to be represented a single time. As can be seen, this means a significative improvement for all datasets (except for geonames), saving up to 60.74% of the terms for dbtune; and ii) the proportion of predicates is always a less significative fraction of the dictionary.

### 5.2 Analyzing Compressed String Dictionaries for RDF

This evaluation is carried out on the dictionaries obtained from the datasets studied above. We analyze space/time tradeoffs for each technique and for URI, Bnode, and Literal dictionaries. Plain (referred to as RG [16]) and compressed (referred to as RRR [23]) bitmaps are studied in these tests.

The Hash technique reserves a table with an overhead of 10% (\( n = 1.1 * m \)) and compacts it by using a bitmap RG configured with sampling 20. Tests performed on other load factors report comparable results. PFC and HTFC techniques are configured on different bucket sizes: \( b = 2^x \), for all \( x \in [1,10] \). Thus, we obtain results for buckets containing from \( 2^1 \) to \( 2^{10} \) terms. Finally, the FMI technique is implemented by using plain (FMI-RG) and compressed (FMI-RR) bitmaps. FMI-RG is parameterized with sampling values \( s = \{4,20,40\} \), and FMI-RR with \( s = \{16,64,128\} \).

Compression. Table 2 summarizes the compression results achieved for the datasets. Compression ratios are calculated with \( c_{comp}/c_d \), where \( c_{comp} \) and \( c_d \) are the compressed and the original dictionary sizes respectively. This table organizes the results for URIs, bnodes (only dbtune uses them), and literals. We give the best and the worst ratios for all parameterizable techniques. Note that, for the FMI technique, the FMI-RR variant always obtains the most compressed representations (for \( s = 128 \) and FMI-RG the worst ones for \( s = 4 \)). The well-known compressor gzip is also considered as a reference of our compression achievements.
Results for URI vocabularies give a clear situation. On the one hand, Hash achieves a poor compression: of around 80% of the original raw size. This result is mainly due to Huffman code performs a character-based compression and it cannot take advantage of longer-range correlations existing between the terms in the vocabulary. This discourages its use for large URI vocabularies. On the other hand, HTFC obtains the best ratios for all datasets because the Front-Coding algorithm is able to detect the long common prefixes shared by the terms. HTFC outperforms PFC thanks to the Hu-Tucker compression. Both maximize their effectiveness for increasing bucket sizes. As can be seen, the HTFC representations take between 20.08% (for dbtune) and 5.04% (for uniprot) of the raw size. In the latter case, it even surpasses the gzip effectiveness. This is a very significant achievement because it demonstrates that these techniques can represent the vocabulary in a space close to that used by a universal compressor and also answer locate and extract. However, FMI and PFC are less effective in this scenario. This analysis can be extended to the Bnodes scenario.

A less clear situation arises for Literals. As can be seen, HTFC is the best choice for geonames and wikipedia, whereas FMI is the most effective for the other datasets. However, the FMI effectiveness is the most uniform one. Experiments show that FMI-RRR largely outperforms FMI-RG, and larger sampling values improve compression in both cases. In turn, PFC and Hash obtain poor results for literals. This fits our initial expectations: literal vocabularies show less regularities than URIs or bnodes, and their compression is greatly complicated. The gzip effectiveness verifies this fact: its better ratios are always greater than 20%, whereas this was the upper limit for URIs.

These results show that URIs and bnodes can be highly compressed and HTFC is the most effective choice. However, optimizations for literals are more complicated because they can contain any type of information, and prefix-based compression is not always sufficient. FMI-RRR arises as an interesting solution for literals, outperforming HTFC in some cases. The classic Front-Coding (PFC) achieves a limited success for URI compression, whereas Hashing is clearly discouraged when compact representations are required.

Querying. We design specific micro-benchmarks for testing querying operations: i) locate is studied through a batch of 10,000 terms randomly chosen for each vocabulary, and ii) another batch containing 10,000 random IDs are used for extract. We run 50 independent executions of each batch and average total times to isolate our measurements of external events. These averaged times per batch are then divided by the number of queries (10,000) to obtain the times per query finally reported in the graphics below.

Figure 2 compares locate (left) and extract (right) performance for the URI (up) and literal (bottom) vocabularies from dbpedia. Each graphic draws compression ratios on the X axis and querying times (in $\mu$s/query) on the Y axis (logscale). All the conclusions below can be extended to the other datasets in the current setup, but their graphics are not shown due to lack of space.

All graphics share a general result: the space/time trade-offs for Hash are never the best choice, neither for compression nor at querying times. The results reported for URIs are very clear: PFC always outperforms HTFC in querying because the latter pays the price of Hu-Tucker decompression. However, as commented above, PFC pays a spatial overhead with respect to HTFC. Its compression ratio is 8 percentage points better than that obtained by PFC, but its temporal improvement is less than 20 $\mu$s/query. Thus, HTFC is well-suited for scenarios focused on compression, but PFC is the better choice if spatial requirements are relaxed. Finally, FMI performance is not competitive for URIs.

The analysis for literals is quite complex. PFC achieves excellent times (5 – 10$\mu$s/query), but its space is up to 3 times...
larger than that used by the most effective technique: FMI-RRR. In turn, HTFC largely improves FPC compression, but querying times evolve to 10 – 60 µs/query for competitive tradeoffs. Finally, FMI takes between 200 and 300 µs per extraction and location respectively. Thus, FMI-RRR must be chosen for optimizing space and FPC may be the choice in scenarios where time prevails. However, as explained in [10], FMI is still the only possible choice when more sophisticated queries (such as substring-based queries) are desired. This could be accomplished with a line of future work devised for filter resolution.

5.3 \(D_{\text{comp}}\) Performance

As explained above, two functional configurations for \(D_{\text{comp}}\) are studied. We choose parameters optimizing the desired dimension within a competitive space/time tradeoff:

- \(D^{(C)}_{\text{comp}}\) is optimized for compression. It implements URI and bnode dictionaries on HTFC \((b = 16)\), and represents literals by using FMI-RRR with sampling 128.
- \(D^{(Q)}_{\text{comp}}\) is optimized for querying. It implements URI and bnode dictionaries on FFC \((b = 8)\), and represents literals by using FMI-RRR with sampling 4.

<table>
<thead>
<tr>
<th>Dict.</th>
<th>(s_p) (MB)</th>
<th>(\text{RDF3X})</th>
<th>(D^{(C)}_{\text{comp}})</th>
<th>(D^{(Q)}_{\text{comp}})</th>
</tr>
</thead>
<tbody>
<tr>
<td>geneshas</td>
<td>133.09</td>
<td>193.47%</td>
<td>27.70%</td>
<td>50.94%</td>
</tr>
<tr>
<td>wikipedia</td>
<td>295.00</td>
<td>177.64%</td>
<td>29.37%</td>
<td>55.27%</td>
</tr>
<tr>
<td>uniprot</td>
<td>734.33</td>
<td>152.12%</td>
<td>27.61%</td>
<td>45.21%</td>
</tr>
<tr>
<td>dbtune</td>
<td>984.66</td>
<td>142.34%</td>
<td>21.99%</td>
<td>41.68%</td>
</tr>
<tr>
<td>dnpedia</td>
<td>5213.76</td>
<td>115.96%</td>
<td>80.92%</td>
<td>64.11%</td>
</tr>
</tbody>
</table>

Table 3: Compression results for RDF dictionaries.

Table 3 shows compression effectiveness for \(D_{\text{comp}}\). We include the sizes of the dictionaries used in RDF-3X [22] to compare our results with respect to a real-world solution (note that we measure the space that \(D_{\text{comp}}\) takes in memory, but RDF-3X size is measured on disk, so additional space is required to be loaded in memory). As can be seen, \(D^{(Q)}_{\text{comp}}\) configuration takes approximately twice the space used by \(D^{(C)}_{\text{comp}}\). The broad difference existing between them allows for some other configurations whose size can be tuned in accordance to specific application requirements. The comparison of our two variants with respect to RDF-3X gives a magnitude of our achievements with regard to the representation of RDF dictionaries. Whereas RDF-3X always uses more space than the original raw dictionary (remember that it combines two data structures for the dictionary), our worst \(D^{(C)}_{\text{comp}}\) result (for dnpedia) uses 64.11% of the original space, while the best one for \(D^{(Q)}_{\text{comp}}\) is only a 30.32%. Thus, \(D_{\text{comp}}\) reduces the space taken by RDF-3X between 2 and 7 times for the studied datasets.

These results guarantee that \(D_{\text{comp}}\) can be tuned to achieve highly-compressed dictionaries. This saves processing resources and enables larger size dictionaries to be managed in a fixed main memory, but also achieves very efficient querying performance. It is studied through a heterogeneous set of real-world dnpedia queries from the log of the USEWOD/2011 Challenge\(^4\). We design a batch of 10,000 queries chosen at random, and execute it in 50 independent repetitions. Times reported are averaged following the procedure used in the previous experiments.


Figure 3: locate (top) and extract (bottom) times.

Figure 3 shows locate and extract times. As can be seen, \(D^{(C)}_{\text{comp}}\) always outperforms \(D^{(Q)}_{\text{comp}}\). It is worth noting that times obtained by our two variants are always less than 10µs per query except for literals. In this case, the use of a more general representation (like FMI) slightly reduces the performance achievable through the other techniques. Note that extract is faster than locate in all cases. Thus, a better performance is achieved for the most used operation in SPARQL engines. The RDF-3X performance is also analyzed. We run the query batch and measure the time that it uses for locate and extract in two different scenarios: “cold” (no data is preloaded in the system main memory) and “warm” (the required data are available in the main memory). The comparison is unfair in the cold scenario because RDF-3X needs data to be transferred from disk; these operations are performed in some milliseconds (one order of magnitude above our technique). As can be seen in figure 3, the test in the warm scenario reduces the times to the level of microseconds, but it never improves our approaches for locate, and only surpasses \(D^{(Q)}_{\text{comp}}\) for extract of literals. However, in this latter case, RDF-3X is unable to handle tagged literals, whereas our approaches give specific support for them.

6. CONCLUSIONS AND FUTURE WORK

This paper addresses compressed representations for RDF dictionaries. We apply existing techniques for string dictionaries to the specific case of RDF and obtain simple compressed representations for URI, blank node and literal dictionaries. This experience is integrated within a novel compressed technique, called \(D_{\text{comp}}\), which compressed the original dictionary up to 22 – 64% of its original size and answered queries in 1 – 50µs. These results represent an improvement on the state-of-the-art (studied through dictionaries
modeled in RDF3X), \( D_{\text{comp}} \) i) uses between 2 – 7 times less space, and ii) answers queries in more efficient time for all cases except for literal extraction. However, \( D_{\text{comp}} \) gives advanced support for managing literals tagged by language or datatype.

Our future work firstly focuses on integrating \( D_{\text{comp}} \) as a dictionary index within an existing SPARQL engine and testint both space and time improvements over the current solutions. Besides, the use of \( D_{\text{comp}} \) within a SPARQL engine provides interesting features for filtering. Our main line of future work is to optimize these features for early filter evaluation. We are working to support more advanced operations on \( D_{\text{comp}} \). Prefix-based searches are easily implementable for PFC and HTFC, and general substring matching can be supported in FM [10]. Note that this latter feature is essential for the expressive regex filtering. Achieving this goal can also influence SPARQL querying performance by integrating early resolution on physical optimization plans.

An additional line of future work focuses on evolving \( D_{\text{comp}} \) to support dynamic operations of insert, delete, and update. These are essential to integrate \( D_{\text{comp}} \) in semantic databases in which dictionaries evolve according to triples management.

Acknowledgments

This work was funded by MICINN (TIN2009-14009-C02-02) and the Millennium Institute for Cell Dynamics and Biotechnology (ICDB) (Grant ICM P05-001-F). The second author holds grants from Erasmus Mundus, the Regional Government of Castilla y León (Spain) and the European Social Fund. NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program. We especially thank to Nieves R. Brisaboa, Francisco Claude and Gonzalo Navarro by their advices about compressed string dictionaries, and to Claudio Gutierrez by his continued support and his magistral lessons about the Web of Data. We also thank to the Database Lab (Univ. of A Coruña, Spain) for lending us its servers for our experiments.

7. REFERENCES