Querying RDF Dictionaries in Compressed Space

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 Computing and
Information Systems (CIS)
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 compacts the dataset and mitigates the scalability issues underlying to its management. 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: $\mathcal{D}_{comp}$, which can be tuned to represent the dictionary in compressed space $(22 - 64\%)$ and to perform basic lookup operations in a few microseconds $(1 - 50\mu s)$. In addition, we propose $\mathcal{D}_{comp}$ as a basis for specific SPARQL query optimizations leveraging its ability for early `FILTER` resolution[1].

Categories and Subject Descriptors

E.4 [**Coding and Information Theory**]; H.3.1 [**Information Storage and Retrieval**]: Dictionaries

General Terms

Algorithms, Design, Performance

Keywords

RDF Dictionaries, Scalability, Compression, SPARQL

1. INTRODUCTION

Nowadays, the so-called **Web of Data** materializes the basic principles of the *Semantic Web* [9]. 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. *Performance* and *scalability* arise as major issues in this scenario and their resolution is closely related to the efficient storage and retrieval of semantic data. Both issues must be analyzed under the World Wide Web Consortium (W3C) Recommendations of the **RDF**[2] (*Resource Description Framework*) data model for conceptual description and **SPARQL**[3] querying language.

RDF provides a graph-based model for structuring and linking data which describes facts of the world [10]. 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**) which maps the strings representing the terms and the integer values (IDs) which identify them. Thus, a dictionary must provide at least two complementary operations: (i) the *string-to-ID* which returns the ID of a given string, and (ii) the *ID-to-string* which retrieves the string identified by a given ID. Both operations are exhaustively used by SPARQL engines during the query resolution 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, and also enables great simplifications for the query processor, which can now perform on the ID-triples representation [26].

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 open issue. 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.
- A practical deployment of how compressed string dictionaries are used for representing RDF vocabularies.

[1]This work is based on an earlier work: SAC '12 Proceedings of the 2012 ACM Symposium on Applied Computing, Copyright 2012 ACM 978-1-4503-0857-1/12/03.
http://doi.acm.org/10.1145/2245276.2245343.

- A configurable technique (called $\mathcal{D}_{comp}$) which achieves highly-compressed RDF dictionaries and very efficient performance for basic lookup operations.

- A basic method leveraging $\mathcal{D}_{comp}$ features for early `FILTER` resolution in SPARQL.

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 $\mathcal{D}_{comp}$) is explained in Section 4, providing specific details on lookup and filtering operations. Then, Section 5 shows an empirical study of RDF dictionaries and analyzes the $\mathcal{D}_{comp}$ 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 $\mathcal{D} = \{s_1, s_2, \ldots, s_n\}$ contains all different strings (*vocabulary*) representing the terms used in a dataset. Its basic management is reduced to the efficient resolution of two queries: `locate(`s_i`)` which maps the string s_i into its ID in $\mathcal{D}$ (*string-to-ID*), and `extract(i)` which returns the string s_i identified as i in $\mathcal{D}$ (*ID-to-string*).

Classical approaches for string dictionaries, like *hashing* [13], use much space. It dissuades applications handling large vocabularies (for instance, those running in the Web of Data) from using it, because of the limited size of available memory. The use of *B-tree* [7] 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.* [8] propose a cache-oblivious string B-tree performing Front-Coding [30] compression in the leaves. It is improved by the compressed *permuterm* [18], 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.* [11], 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: (§2.1) *Hashing* as representative of solutions traditionally used for managing string dictionaries; (§2.2) *Front-Coding*, because it excels for representing vocabularies in which long common prefixes are shared between many strings; and (§2.3) *Self-indexes* because they arise as a competitive choice for achieving compressed indexes of any kind of general string collection.

All these techniques are shown by following their original description [11]. Thus, we regard the dictionary as a text $\mathcal{T}_{dict}$ which concatenates all strings ended by a special \$ symbol (it is, in practice, the ASCII zero code).

2.1 Hashing

Hashing [13] 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, but it has no primitive mechanism to answer `extract`. Besides, hashing does not achieve compression by itself: i) it needs space for storing all m strings in the dictionary, and ii) extra storage space is required for representing the hash table itself $H[1, n]$. The concept of *load factor*: m/n $(m < n)$ is worth noting because it influences the space usage and the time performance.

We consider the technique named *HashB-dh* in [11] (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 $\mathcal{T}_{dict}$ with Huffman [22] and performs the hash function over the compressed strings.

Besides, this technique integrates a primitive resolution for `extract`. It sorts $\mathcal{T}_{dict}$ 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 [30] is a technique traditionally used for compressing lexicographically sorted dictionaries. It leverages that consecutive strings are likely to share a common prefix and they can be *differentially* encoded with respect to their preceding string. This differential encoding concatenates, for each string, the integer representing the common prefix length and the substring which represents the remaining suffix.

Front-Coding partitions the sorted dictionary into buckets of b strings to allow efficient searching: in each bucket, the first string is explicitly stored, whereas the other $b - 1$ ones are differentially encoded according to the method previously explained. Given a string in the `locate` operation, a *fast bucket location* is first performed, comparing the first string of each bucket (*e.g.* through a binary search). Then, the location of the string within the correct bucket requires decoding the differences. It is easily implemented by performing a sequential bucket scan, which is also used in the `extract` operation. In this last case, the process locates the bucket representing the given ID: $\lfloor ID/b \rfloor$, and then decodes all strings until the required one. This decodification performance depends on the value of b, hence it allows different space/time tradeoffs to be yielded.

The *Plain Front-Coding* (`PFC`) technique [11] uses VByte [29] for encoding the differential representation of each string (both the prefix length and the remaining suffix). This decision allows all queries to be completed by exclusively running fast bytewise operations. Focusing on spatial effectiveness, the *Hu-Tucker Front-Coding* (`HTFC`) compresses the byte-stream with Hu-Tucker [23] and performs all op-

erations over this compressed representation. This decision allows significative size reductions to be achieved at the price of slightly increasing querying times.

These techniques, performed on a lexicographic $\mathcal{T}_{dict}$ ordering, are especially suitable for vocabularies containing strings with long common prefixes, such as the case of the URIs contained in RDF datasets.

2.3 Self-Indexing

Self-indexes [25] 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 [17] is a self-index modeling the text on the Burrows-Wheeler Transform (`BWT`) [12]. 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 (`FMI`) based approach is proposed in [11] for compressing string dictionaries. It also performs on a lexicographic $\mathcal{T}_{dict}$ ordering and achieves effective compressed representations for all studied scenarios at the price of a less competitive performance for `locate` and `extract`.

In the current scenario, RDF literals are an example of general texts in which any kind of knowledge (natural language summaries, biological sequences, geographic information, among many other types) can be represented.

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, so these are the identifiers used for data integration in the Web of Data. Many terms in this 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 bnode naming convention of N3.
- **Literals** (`L`) can be considered as "end nodes" in RDF graphs because they exclusively 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 classes can take the role of an object. Thus, `(s,p,o)` $\in (U \cup B) \times (U) \times (U \cup B \cup L)$.

SPARQL and RDF Dictionaries. The Web of Data popularity is the basis for the development of RDF management systems (*RDF stores*) that provide efficient storage and lookup infrastructure. As previously explained, dictionaries are used for compression purposes, but their representation is also an issue for querying: SPARQL engines make use of dictionary indexes, in conjunction with evaluation and histogram indexes, for *physical optimization* [20].

SPARQL resolution and RDF dictionaries are clearly related. SPARQL considers *triple patterns* (*i.e.* RDF triples `(s,p,o)` in which `s`, `p`, or `o` may be a variable) as atomic queries for building more complex ones. Thus, the engine 1) **locates** the IDs associated to the terms provided in the triple patterns; 2) the transformed query is performed, and the resulting ID values are bound to the variables given in this query; and 3) the final result is obtained by **extracting** the terms associated to these resulting IDs. This basic process 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` is overused in comparison to `locate` and should be optimized.

State-of-the-art. Many real-world RDF stores, such as the C-store based one [5], Hexastore [28], or RDF-3X [26] among others, maintain dictionary indexes. However, none implements optimized dictionary solutions. All of them use two independent structures, giving ineffective solutions which double the space required for representing the dictionary. B^+-tree disk-oriented based solutions map terms to IDs for efficient `locate` resolution. In some cases, Front-Coding compression is performed on the leaves. For `extract`, structures supporting constant-time direct access (arrays or memory-mapped files) are used for mapping IDs to terms.

Dictionaries are also a core element for RDF exchanging. 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-naive HDT representation (without any kind of compression) reduces the dataset size up to *fifteen* times [16].

Some other specific applications within the Web of Data demand efficient RDF dictionaries. We emphasize its use in *reasoning applications*. In this scenario, Hogan [21] claims that a dictionary of URIs requires a prohibitive amount of memory to be stored and its compression would help to increase the in-memory capacity. It is very relevant for our objectives to consider that in-memory representations can handle a higher degree of reasoning.

4. OUR APPROACH: $\mathcal{D}_{COMP}$

This section describes our approach (referred to as $\mathcal{D}_{comp}$) for representing and querying compressed RDF dictionaries. First, we set up its structural organization (§4.1) as the basis for our compression purposes. We describe, in (§4.2), the `locate` and `extract` operations over the proposed organization. Finally, we show that the $\mathcal{D}_{comp}$ organization also enables some kinds of SPARQL filtering (§4.3).

Running Example. Our explanation is guided by a running example which illustrates how the vocabulary from an RDF excerpt is modeled using $\mathcal{D}_{comp}$. As shown, it comprises 11 triples stating that the *"Symposium on Applied*

```
<http://dbpedia.org/resource/Symposium_on_Applied_Computing>   <http://www.w3.org/2000/01/rdf-schema#label>      "Symposium on Applied Computing" .
<http://dbpedia.org/resource/Symposium_on_Applied_Computing>   <http://www.myexample.org/ontology/edition>       :_blank_node_edition_2012 .
:_blank_node_edition_2012                                      <http://www.myexample.org/ontology/date>          "2012-03-26"^^<http://www.w3.org/2001/XMLSchema#date> .
:_blank_node_edition_2012                                      <http://www.myexample.org/ontology/venue>         <http://dbpedia.org/resource/Riva_del_Garda> .
<http://dbpedia.org/resource/Riva_del_Garda>                   <http://www.w3.org/2000/01/rdf-schema#label>      "Riva del Garda"@en .
<http://dbpedia.org/resource/Riva_del_Garda>                   <http://www.w3.org/2000/01/rdf-schema#comment>    "Riva del Garda is a town and comune in the northern Italian..."@en .
<http://dbpedia.org/resource/Riva_del_Garda>                   <http://www.w3.org/2000/01/rdf-schema#comment>    "Riva del Garda è un comune della provincia di Trento..."@it .
<http://dbpedia.org/resource/Riva_del_Garda>                   <http://dbpedia.org/ontology/country>             <http://dbpedia.org/resource/Italy> .
<http://dbpedia.org/resource/Riva_del_Garda>                   <http://dbpedia.org/ontology/populationTotal>     "15151"^^<http://www.w3.org/2001/XMLSchema#integer> .
<http://dbpedia.org/resource/Riva_del_Garda>                   <http://www.w3.org/2003/01/geo/wgs84_pos#long>    "10.8500003"^^<http://www.w3.org/2001/XMLSchema#float> .
<http://dbpedia.org/resource/Riva_del_Garda>                   <http://www.w3.org/2003/01/geo/wgs84_pos#lat>     "45.88333511"^^<http://www.w3.org/2001/XMLSchema#float> .
```

Running example: RDF excerpt describing the "`Symposium on Applied Computing`" and "`Riva del Garda`".

```
<http://dbpedia.org/ontology/country>
<http://dbpedia.org/ontology/populationTotal>
<http://dbpedia.org/resource/Italy>
<http://dbpedia.org/resource/Riva_del_Garda>
<http://dbpedia.org/resource/Symposium_on_Applied_Computing>
<http://www.myexample.org/ontology/edition>
<http://www.myexample.org/ontology/date>
<http://www.myexample.org/ontology/venue>
<http://www.w3.org/2000/01/rdf-schema#label>
<http://www.w3.org/2000/01/rdf-schema#comment>
<http://www.w3.org/2003/01/geo/wgs84_pos#long>
<http://www.w3.org/2003/01/geo/wgs84_pos#lat>

:_blank_node_edition_2012

"10.8500003"^^<http://www.w3.org/2001/XMLSchema#float>
"15151"^^<http://www.w3.org/2001/XMLSchema#integer>
"2012-03-26"^^<http://www.w3.org/2001/XMLSchema#date>
"45.88333511"^^<http://www.w3.org/2001/XMLSchema#float>
"Riva del Garda"@en
"Riva del Garda is a town and comune in the northern Italian..."@en
"Riva del Garda e un comune della provincia di Trento..."@it
"Symposium on Applied Computing"
```

Figure 1: Vocabulary for the running example.

Computing was held, in 2012, in Riva del Garda", and providing some basic information about this beautiful town. We select the corresponding triples from *DBpedia*, and we also add one bnode `:_blank_node_edition_2012` for linkage purposes. The vocabulary for this excerpt is shown in Figure 1. It contains 21 terms: 12 URIs, 1 bnode, and 8 literals.

4.1 Structural Organization

An RDF dictionary technique must be optimized from two correlated perspectives: i) the space used for its representation, and ii) the time required for answering `locate` and `extract`. On the one hand, the spatial perspective is related 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.

$\mathcal{D}_{comp}$ considers a specific organization combining these two perspectives. A role-based partitioning is firstly considered and all terms in the dictionary are organized according to the role they play in the dataset:

- **Common subjects and objects** (`SO`) organizes all terms which play subject and object roles in the dataset. Thus, this partition allows these terms to be represented once, achieving significative spatial savings (as shown in Section 5.1, up to 60% of the strings may be in `SO`). Terms playing as subject and object are mapped to the range `[1, |SO|]`.
- **Subjects** (`S`) organizes all subjects which do not play an object role. They are mapped to `[|SO|+1, |SO|+|S|]`.
- **Objects** (`O`) organizes all objects which do not play a subject role. They are mapped to `[|SO|+1, |SO|+|O|]`.
- **Predicates** (`P`) maps all predicates to `[1,|P|]`.

This role-based organization allows $\mathcal{D}_{comp}$ to perform on three ID-range mapping terms in `[1, |SO|+|S|]` (*subjects*), `[1, |SO|+|O|]` (*objects*), and `[1, |P|]` (*predicates*). Although a given ID can belong to different ranges, ambiguity issues cannot arise in `extract` because the general role (subject, object or predicate) is always known and can be provided together with the term ID.

Each partition is then subdivided by attending to the classes (U, B, L) that they can store. It allows the technique which best adjusts each class to be chosen in accordance to its features and its application requirements. That is, a specific subdictionary is used to represent the terms for each class within each partition. This implies that each subdictionary handles its specific mapping, Thus, each term is locally identified within its corresponding subdictionary.

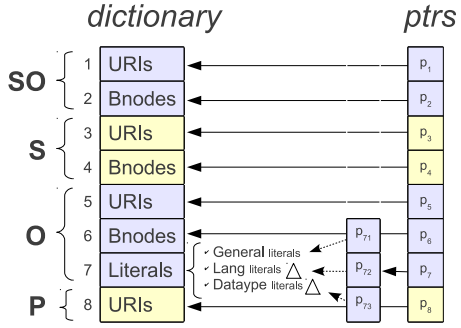

Figure 2: $\mathcal{D}_{comp}$ organization (dictionary+ptrs).

Figure 2 illustrates the resulting organization. As can be seen, the partitions `SO` and `S` are split into URIs and bnodes. Moreover, `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. Tagged-literals are divided again to classify all different languages and datatypes used in the dataset. This hierarchy delimits the ranges for a given language or datatype, allowing string tags to be removed in the final literal representation. Finally, the partition `P` only contains URIs.

Figure 3 shows the $\mathcal{D}_{comp}$ organization for the RDF excerpt

described in the running example. Note that bnode subdictionaries are empty in non-shared subjects and objects.

As stated, each subdictionary owns a local mapping and terms are locally identified within them. However, the RDF graph after ID replacemen (that is, the ID-triples) must be unambiguously represented through global IDs. Thus, $\mathcal{D}_{comp}$ has to implement a mechanism for translating global and local IDs. This mechanism, referred to as *ptrs*, is a very small multi-level mapping structure (shown in Figure 2 and in practice in Figure 3). Each cell in *ptrs* stores two elements: 1) a pointer to the corresponding class subdictionary, and 2) an integer value representing the number of terms previously organized in the corresponding role. That is, the i^{th} cell in the first level of *ptrs* stores the value $ptrs[i] = ptrs[i-1]+t_{i-1}$, where t_{i-1} is the number of terms organized in the subdictionary $i-1$. Some exceptions must be considered:

- $ptrs[1] = 0$ because $\mathcal{D}_{comp}$ forbids any term from being represented before the URIs in the partition `SO`.
- $ptrs[8] = 0$ because predicates are identified within their exclusive range of IDs.
- $ptrs[5] = ptrs[3] = ptrs[2] + t_2$ because both cells store the number of terms represented in the partition `SO`. For instance, in the running example (Figure 3), $ptrs[5] = ptrs[3] = 2$ because there are two previous terms in the `SO` partition.

The second-level of *ptrs* is required for managing the literal subpartitions in `O`. Three additional cells are used:

- The first cell points to the general literals subdictionary and stores the value $ptrs[7,1] = ptrs[7]$.
- The second cell points to the lang-tagged literals representation and stores the value $ptrs[7,2] = ptrs[7,1] + t_{7,1}$, where $t_{7,1}$ is the number of general literals in $\mathcal{D}_{comp}$.
- Finally, the third cell points to the datatype-tagged literals representation and stores the value $ptrs[7,3] = ptrs[7,2]+t_{7,2}$, where $t_{7,2}$ is the number of lang-tagged literals in $\mathcal{D}_{comp}$.

In addition, *ptrs* stores two simple indexes for language-tagged literals, **lang**, and another for datatype-tagged literals, **dtype**. These indexes respectively point to the beginning of each language and datatype subdictionary. They store, respectively, the datatype and language keys allowing them to be deleted in each literal. This decision saves a lot of space because each different tag is represented once.

In the running example (Figure 3), *general literals* stores a single term, whereas there are three *language-tagged literals*: two English (en) and one Italian (it), and four *datatype-tagged literals*: one date, two floats and one integers. As can be seen, all these tags are indexed and represented once.

Transforming local and global IDs. *Ptrs* is the key structure for transforming local IDs into global IDs and viceversa. We introduce a simple notation to explain both translation operation. We refer to l_j the l^{th} local ID in the j^{th} subdictionary. For instance, `blank_node_edition_2012` is identified in the example through the local ID 1 in the 2^{th} subdictionary, and it is named as $\mathbf{1}_2$.

In the first operation, `local-to-global`, a local ID l_j is transformed into its global counterpart as $g(l_j) = l+ptrs[j]$. For instance, `blank_node_edition_2012` is translated to its global ID as: $g(1_2) = 1+ptrs[2] = 1+1 = \mathbf{2}$. An exception occurs for the subdictionary representing literals in the partition `O`. In this case, the process is slightly different:

- If l_j belongs to the subdictionary of *general literals*, the corresponding global ID is obtained as $g(l_j) = l + ptrs[7,1]$.
- If l_j belongs to the subdictionary of *language-tagged literals*, we first look for the language key in the index *lang* and retrieve the value stored for it: $lang[x]$. The global ID is finally obtained as $g(l_j) = l + lang[x]$.
- If l_j belongs to the subdictionary of *datatype-tagged literals*, the translation is performed as in the previous case; the datatype key is found in its index: $dtype[y]$, and the global ID is obtained as $g(l_j) = l + dtype[y]$.

For instance, the term `Riva del Garda is a town and comune in the northern Italian...` is locally identified, in the running example, as 2_7 in the first division (en) of *lang-tagged literals*. In this case, the corresponding global ID is obtained as $g(2_7) = 2 + lang[1] = 2 + 4 = 6$.

The opposite transformation: `global-to-local`, is also implemented over *ptrs*. Given a global ID i, the first step determines the j^{th} subdictionary in which i is represented. The value of j is obtained through the condition: $ptrs[j] < i \leq ptrs[j+1]$. Once j is known, the local value is obtained as $l(i) = i - ptrs[j]$ for the general case. An exception occurs for $j = 7$. In this case, the global ID refers to a literal term and its corresponding subdictionary must be determined in the second level of *ptrs* through the condition: $ptrs[7,k] < i \leq ptrs[7,k+1]$. Three subcases arise in this situation:

- If $k = 1$, the term is a *general literal*: its local ID is obtained as $l(i) = i - ptrs[7,1]$.
- If $k = 2$, the term is a *language-tagged literal* and we must find its language key in the corresponding index. In this case, the key x is determined as $lang[x] < i \leq lang[x+1]$, and the local ID is obtained as $l(i) = i - lang[x]$.
- If $k = 3$, the term is a *datatype-tagged literal* and we must determine its datatype key as in the previous case. The local ID is finally obtained as $l(i) = i - dtype[y]$.

Let us consider the global ID $i = 6$. Comparing the values in the first level of *ptrs*, we verify that it is represented in the 7^{th} subdictionary, because 6 is still greater than $ptrs[7] = 3$. Then, the global ID 6 is a literal term playing as object.

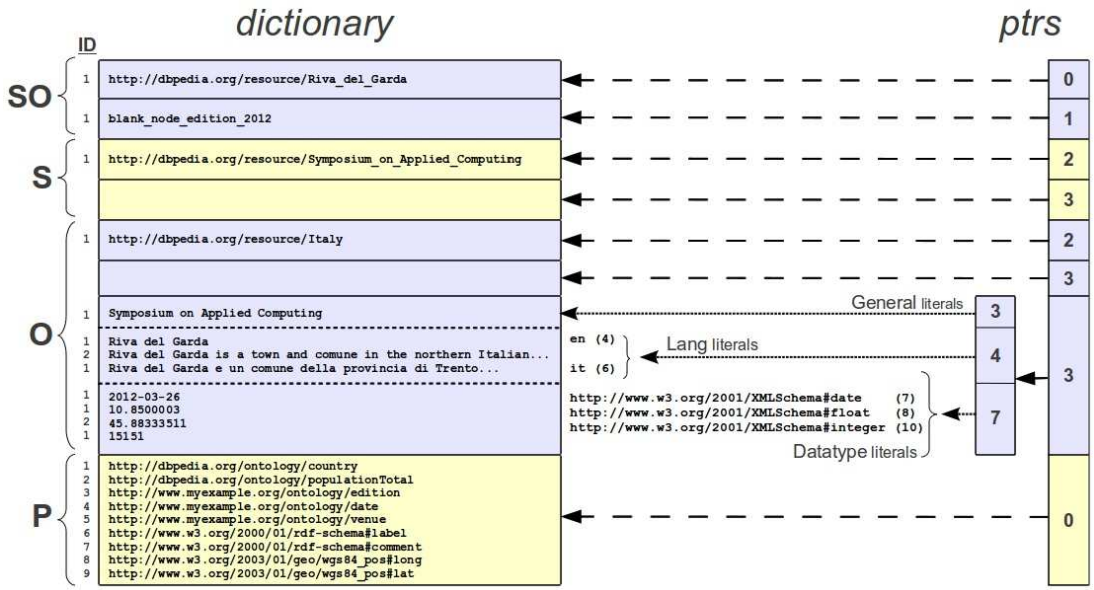

Figure 3: $\mathcal{D}_{comp}$ organization for the RDF excerpt described in the running example.

We now determine the corresponding class of literal in the second-level of *ptrs* and find that it is a *language-tagged literal* because $ptrs[7,2] = 4 < 6 \leq 7 = ptrs[7,3]$. The language is finally looked for in the index lang. It is an English-tagged literal because the ID is represented in the subdictionary related to the first language key: $lang[1] = 4 < 6 \leq 6 = lang[2]$. Finally, the local ID is obtained as $l(6) = 6 - lang[1] = 6 - 4 = 2$, which corresponds to the English literal term `Riva del Garda is a town and comune in the northern Italian...`.

Ptrs Implementation. *Ptrs* is implemented using basic data structures, and its size is negligible for real-world RDF dictionaries. On the one hand, the first two levels of *ptrs* are stored through an array of 11 cells: 8 for the first level, and 3 for the second one. On the other hand, the number of different languages and datatypes modeled in an RDF dictionary depends on the dataset features. However, this number is very small in practice (only several tens, in the worst case), and these indexes can be efficiently implemented through two lexicographically sorted arrays which enable efficient searches for key and global ID.

Finally, it is worth noting tha t any markup symbol is explicitly stored in the corresponding subdictionary. As can be seen, symbols `<` and `>` enclosing URIs, `:_` prefixing bnodes, and quotes delimiting literals are not stored in $\mathcal{D}_{comp}$ because all them can be deducted according to the class of terms represented in each subdictionary. It also allows small spatial savings to be achieved.

4.2 Basic Lookup Operations

The minimal functionality required for an RDF dictionary is reduced to the efficient implementation of two basic lookup operations: `locate` and `extract`. We detail below how $\mathcal{D}_{comp}$ provides this functionality within a generic SPARQL querying process. We consider that a query ($\mathcal{Q}$) comprises one or more triple patterns and each one contains **terms** drawn from URI, bnode and literal classes: $\mathcal{C} = \{U, B, L\}$, and **variables**. Both terms and variables can play the roles of subject, predicate or object: $\mathcal{R} = \{S, P, O\}$. We define $\mathcal{T}$ and $\mathcal{V}$ as the sets which, respectively, contain the terms and variables in $\mathcal{Q}$.

A SPARQL processor firstly parses $\mathcal{Q}$ to obtain the corresponding sets $\mathcal{T}$ and $\mathcal{V}$. Let us suppose that we ask for the descriptive comment of all Italian towns in our running example. This query is expressed as follows:

```
SELECT ?town ?comment
WHERE
{
  ?town <http://dbpedia.org/ontology/country> <http://dbpedia.org/resource/Italy> .
  ?town <http://www.w3.org/2000/01/rdf-schema#comment> ?comment .
}
```

Therefore, the SPARQL processor obtains the set of terms: $\mathcal{T} = \{$`http://dbpedia.org/ontology/country`, `<http://dbpedia.org/resource/Italy>`, `<http://www.w3.org/2000/01/ rdf-schem-a#comment`$\}$, and the variables one: $\mathcal{V} = \{$`?town`, `?comment`$\}$. The next step consists of locating the ID corresponding to each term $t_i \in \mathcal{T}$. It requires many `locate` lookups as terms in the set $\mathcal{T}$. Once all terms are transformed into their corresponding IDs, the SPARQL query is rewritten, and it is run over the ID-triples representation. The query resolution outputs a series of ID values matching for the variables in $\mathcal{V}$. Thus, the last step performs many `extract` operations as results are obtained, for each variable in $\mathcal{V}$, and the corresponding terms are reported within the final query result. In the previous example query, this step extracts the terms binded to the ID-results for the variables `?town` and `?comment`.

We detail below how the location and extraction processes are implemented in $\mathcal{D}_{comp}$ and illustrate them using the example query above.

Locate. This operation implements the translation `string-to-id`. Thus, it requires the term $t_i \in \mathcal{T}$ as key for accessing to $\mathcal{D}_{comp}$. However, it uses two additional properties about

t_i: the role $r_i \in \mathcal{R}$ that it plays in $\mathcal{Q}$, and the class $c_i \in \mathcal{C}$ from which it is drawn. First, the algorithm for `locate` uses the role to determine the partition to be accessed, and then c_i is considered for locating the cell $ptrs[j]$ which stores the information for the dictionary D_j to be queried. `locate(`t_i`)` is performed on D_j, and the ID representing the term: l_j, is returned. However, l_j is a local ID and must be transformed into its global counterpart by using the `local-to-global` method explained above.

Two variants exist in this process. On the one hand, terms playing as subject or object can be represented in the common partition `SO` or in their specific one. It implies that locating a subject or object firstly looks for in the smallest dictionary and if the term is not found, the other one is queried. On the other hand, as stated, tagged literal terms need to make use of their corresponding index to determine the subdictionary representing their language or datatype. In both cases, each the `local-to-global` method is used for translation purposes.

Let us analyze how the terms in our query are located:

- t_1=`<http://dbpedia.org/ontology/country>` is an URI playing the predicate role in the first triple pattern, so the locate operation is invoked with the term t_1, and the parameters $r_1 = P$ and $c_1 = U$. The subdictionary of predicates is queried and the global ID **1** is returned.

- t_2=`<http://dbpedia.org/resource/Italy>` is an URI which plays the object role in the first triple pattern, so the locate operation is invoked with the term t_2, and the parameters $r_2 = O$ and $c_2 = U$. The term is firstly searched in the subdictionary of URI within the partition `SO`, but it is not found. The `locate` query is repeated in the URIs subdictionary in `O`, obtaining the local ID **1**. It is transformed into the corresponding global ID: $1 + ptrs[5] = \mathbf{3}$ which is used for replacement in the original SPARQL query.

- Finally, t_3=`<http://www.w3.org/2000/01/rdf-schema#comment>` is also an URI playing as predicate role in the second triple pattern, so the locate operation is invoked with the term t_3, and the parameters $r_3 = P$ and $c_4 = U$. The subdictionary of predicates is queried again and the global ID **7** is returned.

After these operations, the query is rewritten as: `SELECT ?town ?comment WHERE {(?town 1 3) (?town 7 ?comment)}`, and this is the representation provided for the SPARQL engine. The expected result is the URI of Riva del Garda for the town and the literals, in English and Italian, representing comments about them. These results are returned as follows: `(?town,?comment)={(1,6);(1,7)}`. Thus, the first element in each pair is a result for the variable `?town`, whereas the second element corresponds to results for the variable `?comment`. These IDs are finally transformed into their corresponding terms to return the final result. These translations are performed by using `extract` operations.

Extract. This operation implements the translation `id-to-string`. Thus, it retrieves the term associated to a given global ID: g_x, returned in the result set of a query. Note that g_x is a global ID, so a `global-to-local` transformation is required when a given subdictionary is queried. Let us see how the previous bindings are extracted for each variable involved in the query:

- The variable `?town` is binded to the global ID `1`. In this case, the `global-to-local` operation returns the local ID 1 within the 1^{th} subdictionary, and the corresponding term is extracted in it: `extract(1)`, returning the URI: `<http://dbpedia.org/resource/Riva_del_Garda>`.

- The variable `?comment` is binded to the global IDs `6`, and `7`. The `global-to-local` operation over `6` returns that the ID is represented in the "en"-tagged literals subdictionary, with a local ID 2. Thus, we know that the literal has tag `@en`, and it is extracted from this subdictionary: `extract(2)`. The process is similar for the ID `7`: it is represented in the Italian subdictionary (so it has tag `@it`) with local ID 1, so the literal is extracted as `extract(1)`.

Therefore, the final query result comprises the English and Italian comments about Riva del Garda:

```
<http://dbpedia.org/resource/Riva_del_Garda>, "Riva del Garda
    is a town and comune in the northern Italian..."@en.

<http://dbpedia.org/resource/Riva_del_Garda>, "Riva del Garda
    è un comune della provincia di Trento..."@it.
```

4.3 Filter Resolution

A generic RDF dictionary provides `locate` and `extract` resolution, because it holds the mappings between terms and IDs. However, our dictionary organization enables more advanced SPARQL functionality. This is the case of the `FILTER` clause, which restricts solutions to those for which the filter expression evaluates to true [3].

In the following, we show how unary SPARQL filters can be directly resolved over the proposed dictionary. Two types of SPARQL filtering are distinguished:

- ***Tests*** are used for checking if a query result is drawn from a given term class. Thus, three different tests are available for filtering: `isIRI, isBlank`, and `isLiteral`.

- ***Accessors*** use specific internal term information for filtering. Three different accessors are distinguished:

  - `str`: returns the lexical form of a given term. In practice, this accessor is used for retrieving the string version of the argument passed to it [14].

  - `lang`: returns the language tag of a given literal, if it has one. In other case, it returns an empty string.

  - `datatype`: returns the datatype tag of a given literal. If it is a simple (general) literal, or it is tagged with any language information, `datatype` returns the string tag (`<xsd:string>`).

In both cases, $\mathcal{D}_{comp}$ provides direct filter resolution over the dictionary, allowing the query processor to push-up fil-

ter evaluation. This means, in general, to reduce the number of triples to be explored in the query and thereby to improve the overall query performance when $\mathcal{D}_{comp}$ is used in conjunction with any RDF store in the state-of-the-art. These improvements are highly interesting in real-world scenarios by considering that roughly the 50% of the queries perform any kind of filtering [6].

SPARQL tests. As explained above, these filters rely on checking the term class. We illustrate their implementation by restricting that all bindings for the variable `town` must be URIs:

```
SELECT ?town ?comment
WHERE
{
  ?town <http://dbpedia.org/ontology/country> <http://dbpedia.org/resource/Italy> .
  ?town <http://www.w3.org/2000/01/rdf-schema#comment> ?comment .
  FILTER isURI(?town)
}
```

The traditional non-early test evaluation (also available in $\mathcal{D}_{comp}$) runs the SPARQL query by matching the triple patterns against all triples in the dataset. Then, the result set must be checked, one-on-one, with respect to the filter condition, obtaining the final resultant bindings. Note that, thanks to the organization of $\mathcal{D}_{comp}$, this evaluation of the condition can be performed directly on the IDs, *i.e.*, without the need of extraction of the literal mapped to each ID. This is due to the fact that each partition (subdictionary) only holds a type of term; URIs, Blank nodes or Literals. For instance, in the example query, the resolution of the triple patterns binds the variable `?town` to the ID `1`, and we then verify that `1` is in an URI partition by accessing $ptrs$. In this case, $ptrs[1] = 0 < 1 \leq 1 = ptrs[2]$, thus the ID `1` is in the first subdictionary: URIs playing as `SO`.

In contrast, early test evaluation is resolved in a more efficient way because it leverages the information recorded in the $ptrs$ structure for reducing the space of triples to be explored by the query engine. This evaluation algorithm retrieves, from the first-level of $ptrs$, the ranges of contiguous IDs associated to a given class term, and provides them to the SPARQL engine. This decision allows the engine to only explore the triples represented in this space of possible results, directly discarding all the triples out of these ranges because they do not match the filter condition. In the example query, the filtered variable: `?town`, plays as subject and the filter condition restricts its bindings to URIs. Thus, the space of possible results is first limited to the triples whose subject is identified within the ranges `[1,1]` and `[3,3]`, because these are the ranges assigned to the URIs playing as `SO` and `S` in our running example. These ranges are provided to the engine which only search for matching results in them, and return the ID `1` as a valid result.

SPARQL accessors. These filters extract specific information about the terms. We reformulate the original query to only retrieve comments expressed in English about Riva del Garda:

```
SELECT ?town ?comment
WHERE
{
  ?town <http://dbpedia.org/ontology/country> <http://dbpedia.org/resource/Italy> .
  ?town <http://www.w3.org/2000/01/rdf-schema#comment> ?comment .
  FILTER (lang(?comment) = "en")
}
```

In the traditional non-early evaluation method, again, the query is first run over the full dataset and the result set must be individually checked. However, not all accessors can be resolved without term extraction:

- `str` needs checking the lexical value of each returned ID against the string provided in the filter. Thus, the ID is firstly extracted and then compared with respect to the string in the filter.

- `lang` and `datatype` are resolved without extraction because each ID can be directly compared against the range assigned to the corresponding language or datatype. In this case, the resolution requires querying the second level of $ptrs$ and the indexes $lang$ and $dtype$. In the current query, comments are restricted to those expressed in English, so they must identified in the range `[5,6]` assigned to this language. Considering that two bindings are returned: `{6,7}`, only the first one is a result because `7` is not in the valid range.

The early evaluation algorithm proceeds as in the test case. That is, the ranges of possible results are firstly obtained and the query is exclusively performed over them. This way, the set of returned results is already filtered. In our example query, we firstly access to the $lang$ index and retrieves the range `[5,6]` assigned to the English-tagged literals. This range is provided to the engine which only searches possible results among those triples containing an object ID in this range. In this case, the returned result contains the value `6`, because it is the only binding found in the valid range.

It is worth noting that early resolution of other filters, like `regex` or those based on `arithmetic operations`, remains still opened. On the one hand, `regex` could be addressed leveraging specific features for substring resolution [11]. On the other hand, the case of arithmetic filters seems more difficult by considering that $\mathcal{D}_{comp}$ relies on techniques optimized for string dictionaries, and these operations demand efficient management of numerical values. These topic has been recently studied [15], and the solutions proposed for compact indexing of real numbers are an interesting choice for addressing this issue.

5. EXPERIMENTAL SETUP

This section studies the problem of the 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*[2]; in particular, `geonames` gathers geographic concepts, `dbtune` holds music data and `uniprot` contains biological information mainly focused on proteins; `wikipedia`[3] stands for the English Wikipedia links between pages transformed to RDF, and `dbpedia`[4] is a community effort with the aim of making this type of information semantically available on the Web.

[2] `http://km.aifb.kit.edu/projects/btc-2010/`

[3] `http://labs.systemone.at`

[4] `http://wiki.dbpedia.org/Downloads36`

Table 1: Size comparison (sizes are expressed in GB) and role-based configurations for all datasets.

Dataset	dataset size $\mathcal{O}$	dataset size $\mathcal{C}$	dictionary size: c_d	ID-triples size: c_t
geonames	1.00	0.19	0.14 (73.68%)	0.05 (26.32%)
wikipedia	6.72	0.57	0.30 (52.63%)	0.27 (47.37%)
uniprot	9.11	1.21	0.75 (61.99%)	0.46 (38.01%)
dbtune	9.34	1.37	0.98 (71.53%)	0.39 (28.47%)
dbpedia	33.12	6.96	5.15 (73.99%)	1.81 (26.01%)

triples	elements	SO	S	O	P
9,415,253	5,141,366	1.83%	41.03%	57.14%	0.39×10^{-3}%
47,054,407	8,869,064	17.61%	6.77%	75.62%	0.10×10^{-5}%
72,460,981	14,842,666	43.33%	38.79%	17.88%	0.85×10^{-5}%
58,920,361	16,589,644	60.74%	14.01%	25.24%	0.02×10^{-3}%
232,542,405	67,012,756	24.85%	2.64%	72.45%	0.59×10^{-3}%

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 $\mathcal{D}_{comp}$. It is studied in (§5.3) through two functional configurations: $\mathcal{D}^{(C)}_{comp}$ is focused on compression effectiveness and $\mathcal{D}^{(Q)}_{comp}$ 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 [19]) and *compressed* (referred to as RRR [27]) 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 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 $\mathcal{C} = c_d + c_t$ bytes.

Table 1 (left) shows this comparison. Columns $\mathcal{O}$ and $\mathcal{C}$, 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 $c_d/\mathcal{C}$ and $c_t/\mathcal{C}$ 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 [19]) and *compressed* (referred to as RRR [27]) 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-RRR`) bitmaps. `FMI-RG` is parameterized with sampling values $s = \{4, 20, 40\}$, and `FMI-RRR` 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-RRR` 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

Table 2: Comparison of general techniques for string dictionaries (s_r is the dictionary raw size in MB).

URIs	s_r (MB)	gzip	Hash	PFC	HTFC	FMI
geonames	102.77	9.74%	82.24%	19.17% – 64.10%	**11.87**% – 44.05%	25.09% – 61.29%
wikipedia	212.65	13.62%	77.68%	22.18% – 64.62%	**15.37**% – 45.23%	25.25% – 63.13%
uniprot	520.90	7.20%	80.94%	8.71% – 58.40%	**5.04**% – 42.44%	23.69% – 64.56%
dbtune	281.54	19.47%	78.44%	30.88% – 68.30%	**20.08**% – 48.22%	31.47% – 67.53%
dbpedia	1553.46	14.98%	78.87%	27.46% – 67.16%	**19.82**% – 48.17%	26.97% – 66.12%
Bnodes	s_r (MB)	gzip	Hash	PFC	HTFC	FMI
dbtune	623.73	9.69%	71.14%	22.83% – 64.08%	**14.50**% – 42.66%	20.75% – 72.92%
Literals	s_r (MB)	gzip	Hash	PFC	HTFC	FMI
geonames	32.30	29.44%	121.70%	55.90% – 87.70%	**39.70**% – 71.22%	40.65% – 84.93%
wikipedia	82.36	22.13%	107.49%	41.49% – 78.82%	**29.24**% – 61.78%	34.62% – 80.80%
uniprot	213.43	47.05%	72.85%	90.89% – 97.40%	59.48% – 65.86%	**53.69**% – 79.00%
dbtune	79.39	20.99%	90.37%	70.04% – 89.02%	52.04% – 68.36%	**32.56**% – 87.51%
dbpedia	3660.30	22.87%	74.34%	78.31% – 89.13%	53.78% – 64.01%	**30.29**% – 82.62%

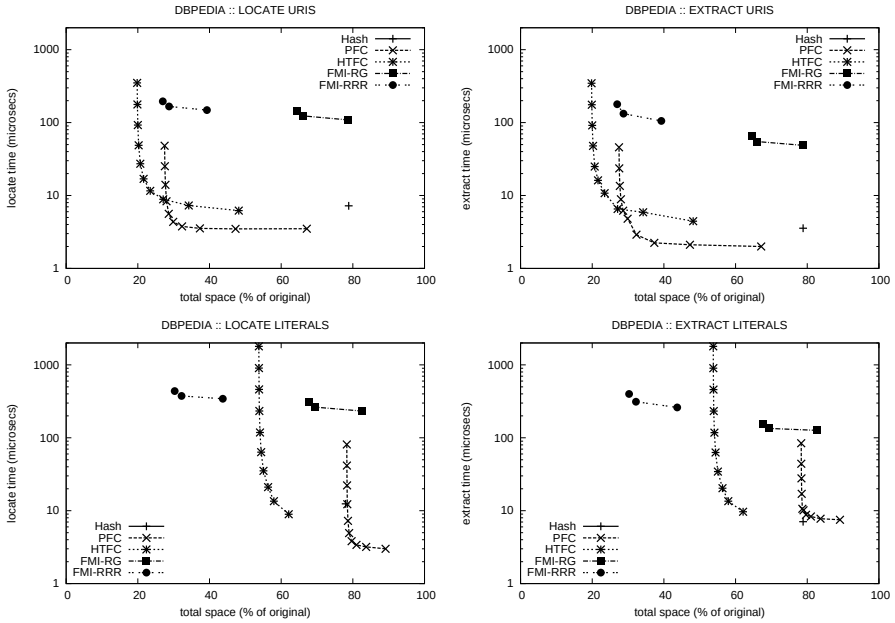

Figure 4: `locate` and `extract` times for *URIs* (top) and *literals* (bottom) from `dbpedia`.

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 effectiveness of `gzip`. 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`. `PFC` and `FMI` 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 effectiveness of `FMI` is the most uniform. 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 effectiveness of `gzip` 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 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 4 compares `locate` (left) and `extract` (right) performance for the **URI** (upper) 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 tradeoffs 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 μs/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 `PFC` 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 `PFC` may be the choice in scenarios where time prevails. However, `FMI` is still the only choice when more sophisticated queries (such as substring-based ones) are desired [11]. This accomplishes with the line of future work devised for filter resolution.

5.3 $\mathcal{D}_{comp}$ Performance

As explained above, two functional configurations for $\mathcal{D}_{comp}$ are studied. We choose parameters optimizing the desired dimension within a competitive space/time tradeoff:

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

Table 3 shows compression effectiveness for $\mathcal{D}_{comp}$. We include the sizes of the dictionaries used in RDF-3X [26] to compare our results with respect to a real-world solution (note that we measure the space that $\mathcal{D}_{comp}$ 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, $\mathcal{D}^{(Q)}_{comp}$ configuration takes approximately twice the space used by $\mathcal{D}^{(C)}_{comp}$. 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 $\mathcal{D}^{(Q)}_{comp}$ result (for `dbpedia`) uses 64.11% of the original space, while the best one for $\mathcal{D}^{(C)}_{comp}$ is only 30.32%. Thus, $\mathcal{D}_{comp}$ reduces the space taken by RDF-3X between 2 and 7 times for the studied datasets.

Table 3: Compression results for RDF dictionaries.

Dict.	s_r (MB)	**RDF3X**	$\mathcal{D}^{(C)}_{comp}$	$\mathcal{D}^{(Q)}_{comp}$
`geonames`	135.07	195.47%	27.09%	50.94%
`wikipedia`	295.00	177.64%	29.37%	55.27%
`uniprot`	734.33	152.12%	27.61%	45.21%
`dbtune`	984.66	142.34%	21.99%	41.08%
`dbpedia`	5213.76	115.93%	30.32%	64.11%

These results guarantee that $\mathcal{D}_{comp}$ 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 `dbpedia` queries from the log of the USEWOD'2011 Challenge[5]. We designed a batch of $10,000$ queries chosen at random, and executed it in 50 independent repetitions. Times reported are averaged by following the procedure used in the previous experiments.

Figure 5 shows `locate` and `extract` times. As can be seen, $\mathcal{D}^{(Q)}_{comp}$ always outperforms $\mathcal{D}^{(C)}_{comp}$. It is worth noting that times obtained by our two variants are always less than $10\mu s$ 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 5, 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 $\mathcal{D}^{(Q)}_{comp}$ for `extract` (literals). However, RDF3X 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 $\mathcal{D}_{comp}$, which compressed the original dictionary up to $22-64\%$ of its original size and answered queries in $1-50\mu s$. These results represent an improvement

[5] `http://data.semanticweb.org/usewod/2011/challenge.html`

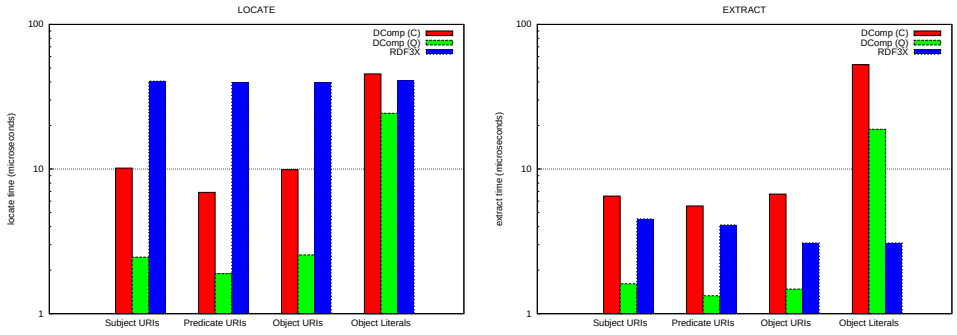

Figure 5: `locate` (left) and `extract` (right) times.

on the state-of-the-art (studied through dictionaries modeled in RDF3X). $\mathcal{D}_{comp}$ i) uses between $2-7$ times less space, and ii) answers queries in more efficient time for all cases except for literal extraction. However, $\mathcal{D}_{comp}$ gives advanced support for managing tagged literals. This experience has been evidenced in an innovative technique providing efficient SPARQL resolution in compressed space [24].

Our future work firstly focuses on integrating $\mathcal{D}_{comp}$ as a dictionary index within an existing SPARQL engine and test in both space and time improvements over the current solutions. Besides, the use of $\mathcal{D}_{comp}$ within a SPARQL engine provides interesting features for filtering. Our main line of future work is to optimize these features for early evaluation. We are also working to support more advanced operations on $\mathcal{D}_{comp}$. Prefix-based searches are easily implementable for `PFC` and `HTFC`, and general substring matching can be supported in `FMI` [11]. 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 $\mathcal{D}_{comp}$ to support dynamic operations of insert, delete, and update. These are essential to integrate $\mathcal{D}_{comp}$ in semantic databases in which dictionaries evolve according to triples management.

Acknowledgments

This work was funded by the MICINN (Spain): TIN2009-14009-C02-02. The second author holds grants from Erasmus Mundus, the Regional Government of Castilla y León (Spain) and the European Social Fund. The third author is now a member of the NICTA which 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 would especially like to thank Nieves R. Brisaboa, Francisco Claude and Gonzalo Navarro for their advice about compressed string dictionaries, and Claudio Gutierrez for his continued support and his magistral lessons about the Web of Data. We would also like to thank the Database Lab (Univ. of A Coruña, Spain) for lending us its servers for our experiments.

7. REFERENCES

[1] *Compact Data Structures Library (libcds).* `http://libcds.recoded.cl/`.

[2] *RDF Primer.* W3C Recommendation. 2004. `http://www.w3.org/TR/rdf-primer/`.

[3] *SPARQL Query Language for RDF.* W3C Recommendation. 2008. `http://www.w3.org/TR/rdf-sparql-query/`.

[4] *Binary RDF Representation for Publication and Exchange (HDT).* W3C Member Submission. 2011. `http://www.w3.org/Submission/2011/03/`.

[5] D. Abadi, A. Marcus, S. Madden, and K. Hollenbach. Scalable Semantic Web Data Management Using Vertical Partitioning. In *Proc. of VLDB*, pages 411–422, 2007.

[6] M. Arias, J. D. Fernández, and M. A. Martínez-Prieto. An Empirical Study of Real-World SPARQL Queries. In *Proc. of USEWOD*, 2011. Available at: `http://arxiv.org/abs/1103.5043`.

[7] R. Bayer and E. E. McCreight. Organization and maintenance of large ordered indices. In *Proc. of ACM SIGFIDET*, pages 107–141, 1970.

[8] M. Bender, M. Farach-Colton, and B. Kuszmaul. Cache-oblivious string B-trees. In *Proc. of PODS*, pages 233–242, 2006.

[9] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American Magazine*, 2001.

[10] C. Bizer, T. Heath, and T. Berners-Lee. Linked Data - The Story So Far. *International Journal on Semantic Web and Information Systems*, 5:1–22, 2009.

[11] N. Brisaboa, R. Cánovas, F. Claude, M. A. Martínez-Prieto, and G. Navarro. Compressed String Dictionaries. In *Proc. of SEA*, pages 136–147, 2011.

[12] M. Burrows and D. Wheeler. A Block-Sorting Lossless Data Compression Algorithm. Technical report, Digital Equipment Corporation, 1994.

[13] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press, 2001.

[14] B. DuCharme. *Learning SPARQL.* O'Reilly, 2011.

[15] A. Fariña, A. Ordóñez, and J. R. Paramá. Indexing sequences of ieee 754 double precision numbers. In *Proc. of DCC*, pages 367–376, 2012.

[16] J. D. Fernández, M. A. Martínez-Prieto, and C. Gutiérrez. Compact Representation of Large RDF Data Sets for Publishing and Exchange. In *Proc. of ISWC*, pages 193–208, 2010.

[17] P. Ferragina and G. Manzini. Opportunistic Data Structures with Applications. In *Proc. of FOCS*, pages 390–398, 2000.

[18] P. Ferragina and R. Venturini. The compressed permuterm index. *ACM Trans. Alg.*, 7(1):art. 10, 2010.

[19] R. González, S. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of rank and select queries. In *Proc. of WEA*, pages 27–38, 2005.

[20] S. Groppe. *Data Management and Query Processing in Semantic Web Databases*. Springer, 2011.

[21] A. Hogan. *Exploiting RDFS and OWL for Integrating Heterogeneous, Large-Scale, Linked Data Corpora*. PhD thesis, National University of Ireland, 2011.

[22] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proc. of the Institute of Radio Engineers*, 40(9):1098–1101, 1952.

[23] D.E. Knuth. *The Art of Computer Programming, volume 3: Sorting and Searching*. Addison Wesley, 1973.

[24] M. A. Martínez-Prieto, M. Arias, and J. D. Fernández. Exchange and Consumption of Huge RDF Data. In *Proc. of ESWC*, pages 437–452, 2012.

[25] G. Navarro and V. Mäkinen. Compressed Full-Text Indexes. *ACM Computing Surveys*, 39(1):art. 2, 2007.

[26] T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *The VLDB Journal*, 19:91–113, 2010.

[27] R. Raman, V. Raman, and S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. of SODA*, pages 233–242, 2002.

[28] C. Weiss, P. Karras, and A. Bernstein. Hexastore: Sextuple Indexing for Semantic Web Data Management. *Proceedings of VLDB Endowment*, 1(1):1008–1019, 2008.

[29] H. Williams and J. Zobel. Compressing Integers for Fast File Access. *The Computer Journal*, 42:193–201, 1999.

[30] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes : Compressing and Indexing Documents and Images*. Morgan Kaufmann, 1999.