# A Compact In-Memory Dictionary for RDF Data

Hamid R. Bazoobandi[1(✉)], Steven de Rooij[1,2], Jacopo Urbani[1,3], Annette ten Teije[1], Frank van Harmelen[1], and Henri Bal[1]

[1] Department of Computer Science, VU University Amsterdam, Amsterdam, The Netherlands
h.bazoubandi@vu.nl, {jacopo,annette,frank.van.harmelen,bal}@cs.vu.nl
[2] Department of Computer Science, University of Amsterdam, Amsterdam, The Netherlands
s.deRooij@uva.nl
[3] Max Planck Institute for Informatics, Saarbrücken, Germany

**Abstract.** While almost all dictionary compression techniques focus on static RDF data, we present a compact in-memory RDF dictionary for dynamic and streaming data. To do so, we analysed the structure of terms in real-world datasets and observed a high degree of common prefixes. We studied the applicability of Trie data structures on RDF data to reduce the memory occupied by common prefixes and discovered that all existing Trie implementations lead to either poor performance, or an excessive memory wastage.

In our approach, we address the existing limitations of Tries for RDF data, and propose a new variant of Trie which contains some optimizations explicitly designed to improve the performance on RDF data. Furthermore, we show how we use this Trie as an in-memory dictionary by using as numerical ID a memory address instead of an integer counter. This design removes the need for an additional decoding data structure, and further reduces the occupied memory. An empirical analysis on real-world datasets shows that with a reasonable overhead our technique uses 50–59% less memory than a conventional uncompressed dictionary.

## 1 Introduction

Dictionary encoding is a simple compression method used by a wide range of RDF [17] applications to reduce the memory footprint of the program. A dictionary encoder usually provides two basic operations: one for replacing strings with short numerical IDs (encoding), and one for translating IDs back to the original strings (decoding). This technique effectively reduces the memory footprint, because numerical values are typically much smaller than string terms. It also boosts the general performance since comparing or copying numerical values is more efficient than the corresponding operations on strings.

Dictionary encoding relies on a bi-directional map, which we call a dictionary, to store the associations between the numeric and textual IDs. If the input


F. Gandon et al. (Eds.): ESWC 2015, LNCS 9088, pp. 205–220, 2015.
DOI: 10.1007/978-3-319-18818-8_13

contains many unique tokens, then the size of the dictionary can saturate main memory and start hampering the functioning of the program. Given the increasing size of RDF datasets, this is becoming a frequent scenario, e.g. [19] reports cases where the size of the dictionary becomes even larger than the resulting encoded data. This becomes particularly problematic for applications that need to keep the dictionary in memory while processing the data.

An additional challenge comes from the dynamic nature of the Web, which demands that the application access and/or updates the dictionary with high frequency (for example when processing high velocity streams of RDF data). This requirement precludes the usage of most existing dictionary compression techniques (e.g. [10,19]) since these sacrifice update performance in order to maximize compression, which was the rational trade-off when processing static RDF data. To the best of our knowledge, there is no method to store dictionaries of RDF data that is space-efficient and allows frequent updates.

The goal of this paper is to fill this gap by proposing a novel approach, called RDFVault, to maintain a large RDF dictionary in main memory. RDFVault design contains *two main novelties*: First, it exploits the high degree of similarity between RDF terms [9] and compresses the common prefixes with a novel variation of a Trie [11]. Tries are often used for this type of problems, but standard implementations are memory inefficient when loaded with skewed data [13], as is the case with RDF [16]. To address this last issue, we present a Trie variation based on a List Trie [7], which addresses the well-known limitations of List Tries with a number of optimizations that exploit characteristics of RDF data.

Second, inspired by symbol tables in compilers, our approach unifies the two independent tables that are normally used for encoding and decoding into a single table. Our unified approach maps the strings not to a counter ID (as is usually the case), but to a memory address from where the string can be reconstructed again. The advantage of this design is that it removes the need of an additional mapping from IDs back to strings.

To support our contribution, we present an empirical analysis of the performance and memory consumption of RDFVault over realistic datasets. Our experiments show that our technique saves 50–59% memory compared to uncompressed hash-based dictionary while maintaining competitive encoding speed and up to 2.5 times slower decoding performance. Given that decoding in a conventional hash-based dictionary is very fast (a single hash table look up), we believe that the decoding speed of RDFVault is still reasonably good, and that in many cases this is a fair price to pay for better memory consumption.

The rest of this paper is organized as follows: In Sect. 2 we report some initial experiments that illustrate the potential saving that we can obtain with redundancy-aware techniques. In Sect. 3 we overview related work on the structure of existing dictionaries, and briefly discuss some of the existing efforts to reduce their memory consumption. In this section we also introduce the Trie data structure which will be the basis of our method. In Sect. 4 we focus on a number of existing Trie variants and analyze their strengths and weaknesses when

applied to RDF data. Then, we present our method in Sect. 5 and an empirical evaluation of its performance in Sect. 6. Finally, Sect. 7 concludes the paper and discusses possible directions for future work.

## 2 String Redundancy in RDF: An Empirical Analysis

It is well known that the hierarchical organization of IRIs produces a large amount of string redundancy in a typical RDF dataset [9]. However, the scale of such redundancy has never been exactly quantified. Therefore, we selected a random subset of four realistic datasets (Table 1 shows more details on each dataset), and for each of them we computed the collective amount of common prefixes of any length.

**Table 1.** Number and type of terms in examined datasets

| Datasets | #Terms (M) | | | #Unique Terms (M) | | | Triples (M) |
|---|---|---|---|---|---|---|---|
| | IRI | Literal | All | IRI | Literal | All | |
| BioPortal [22] | 112.17 | 17.80 | 130 | 3.32 | 4.11 | 7.44 | 43.33 |
| Freebase [12] | 237.59 | 60.50 | 300 | 19.06 | 10.48 | 29.77 | 100 |
| BTC2014 [14] | 228.33 | 20.79 | 300 | 11.47 | 3.42 | 17.97 | 100 |
| DBPedia (EN) [3] | 280.07 | 19.22 | 300 | 50.01 | 1.85 | 51.87 | 100 |

Figure 1(a) reports (divided by type of terms) the results of our experiment. Although we expected some degree of redundancy in RDF data, this level of redundancy was beyond our expectations. We observed that 74–84% of the total space required by all unique strings is occupied by prefixes that appear more than once. Furthermore, the chart also shows that the redundancy is not confined to IRIs, but extends to literals as well (e.g. in Freebase [12] about 80 % of the space is occupied by repeated prefixes of different lengths).

In addition, we calculated the average length of the common prefixes and observed that common prefixes are between 20 to 30 characters long depending on the dataset. Such observations support the idea that minimizing the storage of common prefixes has the potential of effectively reducing the total memory consumption. These findings motivated our research to design an efficient dictionary encoder that exploits these redundancies in RDF strings to reduce the space occupied by the dictionary. The remainder of this paper describes our efforts towards this goal.

## 3 Existing Approaches

*Related work.* Dictionary encoding is a popular technique in real-world applications.(e.g. RDF-3x [21], or Virtuoso [8]). In general the systems that apply dictionary encoding construct the dictionary with two independent data structures,

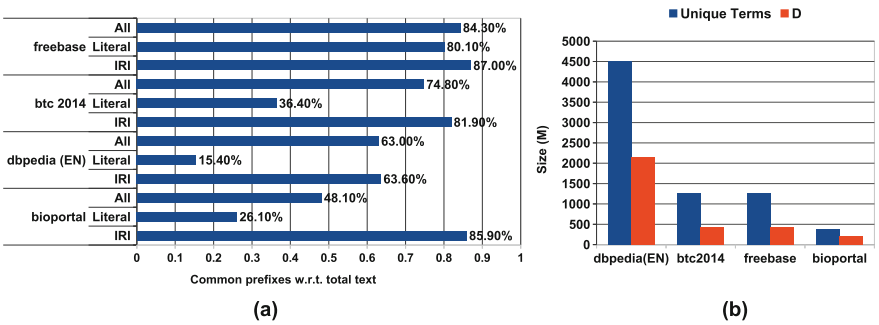


**Fig. 1.** (a) The collective amount of common prefixes of any length categorized by type of terms. (b) The disk space occupied by HDT dictionary versus unique string terms.

one for mapping strings to IDs, and one for mapping IDs to their corresponding strings. The method proposed by [25] tries to alleviate the overhead of common prefixes in IRIs by splitting them based on the last occurrence of '/' character and storing the prefix only once in a separate hash table for all IRIs that share that prefix. The problem with this approach is that in RDF data, common prefixes have a variable lengths. Therefore many common prefixes will be still stored more than once. In addition, our analytical study (see Sect. 2) shows that common prefixes do not occur only in IRIs, but also in literals, which are outside the scope of this optimization.

There are also approaches that offer dictionary compression over static datasets, e.g. HDT [10] applies PPM [6] to compress its D (dictionary) part (*FourSectionDictionary*)[1]. Our experiments presented in Fig. 1(b) compare the disk space occupied by unique strings in datasets presented in Table 1 versus that of HDT dictionary. The Figure clearly shows that in almost all cases, the whole HDT dictionary (strings and IDs) occupies more than 50 % less space than uncompressed strings. Another similar approach [19] is a compact dictionary which partly relies on partitioning terms based on the role they play in the datasets to achieve a dictionary compression level of 20–64%. Although both approaches effectively compact the dictionary, they require the whole dataset to be available at compression time, and they both function under this assumption that the data rarely changes after the dictionary creation. As a result, they support relatively efficient decoding (order of micro seconds in our experiments) but support no new encoding after the dictionary is created. Thus, these techniques are great if the data is static, but inapplicable for dynamic and streaming data sources used in many real-time usecases such as stream RDF processing.

Reference [4] proposes an order preserving in-memory dictionary based on a single data structure that supports dynamic updates, however the approach is vulnerable to memory wastage for highly skewed data with many duplicates (like

[1] https://code.google.com/p/hdt-java/.

RDF data). References [5,24] propose approaches that address the scalability issue of massive RDF compression by resorting to distributed approaches.

*Trie.* If we look at the data structures that are normally used inside the dictionary, then we notice that often $B^+$-trees are chosen if the dictionary is stored on disk, while arrays, hash tables, or memory mapped files are normally preferred if the dictionary is supposed to reside in main memory [19]. Regardless of the data structure, in general existing approaches do not attempt to minimize the storage of common prefixes, and therefore consume significant space.

A Trie [11] (also known as radix or prefix tree) is special multi-way tree that was initially proposed as an alternative to binary trees [15] for indexing strings of variable length. In a Trie, each node represents the string that is spelled out by concatenating the edge labels on the path from the root. The string stored in a Trie is represented by the terminal node, while each internal node represents a string prefix. The children of a node are identified by the character on their edge labels; So, the fastest Trie implementation stores an array of $|\Sigma|$ child pointers in each node, where $\Sigma$ is the alphabet. For instance, if a Trie should store ASCII strings, then the arrays would need to have 128 entries.

To better illustrate the functioning of a Trie, we show in Fig. 2(a) a small example of a standard Trie that supports uppercase English alphabet and contains three simple keys ("ABCZ", "ABCA", and "XYZ"). The example shows that no node in the Trie stores the key associated with that node. Instead, it is the position of the node in the Trie that determine the key associated with that node. In other words, the indices followed to reach a node, determines the key associated with that node. In this example we also see that the strings "ABCZ", and "ABCA" share the part of the Trie that represents the common prefix. Because of this, Tries have the following desirable properties:

- All strings that share the same prefix will be stored using the same nodes. Therefore common prefixes are stored only once;
- Keys can be quickly reconstructed via a bottom up traversal of the Trie;
- Time complexity of insertions, and lookups are proportional only to the *length of the key*, and not to the number of elements in the Trie.

Our experiments in Sect. 2 show that a storage strategy that minimizes string redundancy has the potential of being very effective in terms of resource consumption. Therefore, Tries can potentially be an ideal data structure for the compression of RDF terms in memory. Unfortunately, the most serious drawback of Tries is that if the input is skewed, and the alphabet is large, the Trie nodes become sparse [13] and cause low memory efficiency. In last years, this limitation has received considerable attention, and a number of papers have proposed some interesting solutions. In the next section we discuss the most prominent ones and analyze how they perform in our specific usecase.

## 4 Towards an Optimal Trie Implementation for RDF

*Compact Trie.* In a standard Trie (Fig. 2(a)), each edge represents a single character of the key, and thus all characters of all input strings are represented

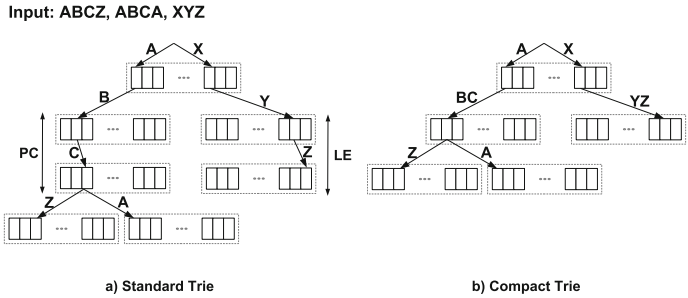


**Fig. 2.** Lazy expansion and path compression optimizations

by pointers between nodes. If the strings represent natural language text, then standard Tries are extremely vulnerable to memory wastage when nodes become sparse [13]. To mitigate this issue, the two following optimizations [20,23] are particularly effective:

**Lazy Expansion.** Chains of single descendant (child) nodes that lead to a terminal node (leaf) are omitted and the eliminated characters are usually stored in the leaf.

**Path Compression.** Single descendent nodes that do not lead to leaves are omitted and the skipped characters are either stored in the (multi-descendant) nodes, or only the numbers of characters is stored in the nodes, and the entire string is stored in the leaves to ensure correctness.

We call a Trie that implements both optimizations a *Compact Trie*. Figure 2(b) shows nodes that are affected by path compression (PC) and lazy expansion (LE) optimizations. Unfortunately, even with these optimizations in place, our experiments (the second column of Table 2) show that still more than 98 % of entries in the pointer arrays of a compact Trie remain unused when we store RDF data. This shows us that even though these two optimizations are useful in the general case, they do not help turn the data structure into a memory-efficient data structure for RDF.

**Table 2.** Percentage of used node pointers in a compact trie and ART when loaded with realistic RDF data.

| Dataset | Compact trie | ART |
|---|---|---|
| BioPortal | 1.58 % | 47.90 % |
| DBPedia (EN) | 1.19 % | 46.60 % |
| Freebase | 1.91 % | 48.03 % |
| BTC2014 | 1.23 % | 44.79 % |

*Burst Trie and HAT Trie.* A Burst Trie [13] is a hybrid data structure comprised of a standard Trie called *access Trie* whose leaves are *containers* that can be any data structure (linked lists by default). HAT-Trie [2] improves performance by using hash tables instead of linked lists. Initially strings are only organized in containers, but once the algorithm detects a container is inefficient, it bursts the container into a Trie node, with multiple smaller containers as its leaves.

An advantage of this hybrid design is that it is more resistant to memory wastage for skewed data. However, this data structure is not attractive for saving common prefixes because (a) it does not minimize the storage of all common prefixes, but only those that are in the access Trie b) the burst Trie does not apply path compression and lazy expansion optimizations, therefore the access Trie can become very inefficient for long strings.

*Adaptive Radix Trie.* Adaptive Radix Tree (ART) [18] further improves memory efficiency, not only by applying the lazy expansion and path compression optimizations, but also by adaptively changing the size of the pointer array in nodes to minimize the number of unused pointers. To this end, ART uses nodes with variable length which grow in size where there is not enough space in their arrays. We measured the effect of this new optimization for RDF data, and report the results in the third column of Table 2. As we can see from the table, the adaptive node policy significantly boosts the memory efficiency compared to the Compact Trie. Nevertheless, still more than half of the pointer array entries are left unused. Therefore, for large Tries with many nodes, the memory efficiency is still unacceptably high.

*List Trie.* The last type of Trie that we considered is the List Trie [7]. This Trie organizes the children pointers of each node in linked lists instead of arrays. The advantage is that, unlike arrays, linked lists are not vulnerable to sparsity. However, the price to pay is that linked lists do not support random accesses. Therefore, in generic cases the performance of a List Trie is significantly lower than other Trie variants. Our experiments (not shown because of space limitations) show that a Standard Trie is more than two times faster than a List Trie in storing English words in a dictionary, though a List Trie consumes 6.3 times less memory than a Standard Trie to do so.

## 5 RDFVault: An In-Memory Dictionary Optimized for RDF Data

In the previous section we analyzed the existing Trie variants and showed why none of them is ideal for RDF. In fact, while Tries remove the problem of string redundancy, array-based tries are still memory inefficient because of the excessive number of unused pointers (Table 2), and list-based Tries cannot guarantee a good performance in generic cases.

To address these limitations, we propose a new variant of a List Trie and use it as an optimized in-memory dictionary named RDFVault for dynamic and streaming RDF data. There are three important factors that differentiate our

solution from existing methods. *First*, our Trie variant uses linked lists (despite their general suboptimality) and improves the performance by introducing a move-to-front policy. *Second*, our dictionary encoding approach removes the need for a dedicated decoding data structure by using as ID the memory location of the Trie node that represents the string. Finally, it further optimizes memory usage by using two different types of nodes in the construction of the Trie. The remainder of this section describe each of these points in more detail.

**Move-to-front policy.** To support our decision to use linked lists, we run an experiment that calculates the distribution of used pointers in a Compact Trie (8-bit characters) loaded with RDF terms of some real-world datasets (Table 1). The results reported in Fig. 3 show that nodes with fewer than four used pointers (out of 256) are the majority. This observation plays in favor of using a list instead of array to keep track of children, because even with $O(n)$ lookup time complexity, when the $n$ is small, the overall cost is reasonably low.

However, using a list is not enough. In fact, the plot shows that there are nodes with more than a dozen children. These nodes, although not many, can constitute a significant performance bottleneck if they appear on frequent paths. For example, if such popular nodes appear during the encoding of a term in the RDF vocabulary, then the overall performance will be severely affected.

Nevertheless, the high skewness and similarities among RDF terms suggest that some nodes in the lists are looked up much more than others. Therefore, to overcome the performance degradation introduced by popular long lists, we introduced a *move-to-front* policy which moves the last accessed child to the beginning of the list. In this way, the popular nodes will tend to move to the front of the list, while the least accessed nodes will automatically drift to the rear. This allows us to have a very compact data structure with no string redundancy, no memory wastage due to unused pointers, and still reasonably fast.

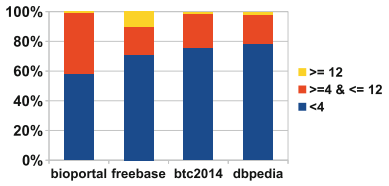


**Fig. 3.** Children distribution of nodes in Tries loaded with RDF data

**Trie as a Dictionary.** The most common approach for dictionary encoding is to assign an ID taken from an internal counter to every distinct term in the input. This requires the dictionary to maintain two maps, one for mapping strings to IDs, and one for inverse operation. In our approach, we improve this by using as ID, the memory address of the Trie node that represents the string. We make this design decision to remove the need for a second decoding data structure, and by doing so, we are actually reducing the implementation of a dictionary

from two maps to a single set which we implement using our memory efficiency Trie. This is possible thanks to this interesting property of Tries (explained in Sect. 3) that offers the reconstruction of strings via a bottom-up traversal in a time complexity proportional to the length of string.

A downside of this choice is that we must add a pointer from every child to its parent to make the upward traversal possible. However, this extra cost is negligible compared to maintaining a whole second data structure in memory.

The main disadvantage of using the memory location as ID is that we are no longer allowed neither to relocate nor to reuse nodes that are associated with strings. This option has a limited impact on deletion operations (we assume that if a string is removed from the dictionary, then the ID assigned to that string is recyclable[2]) but it obliged us to adapt the original insertion algorithm to ensure that this condition is always observed.

In more details, we needed to address two cases: the first is when the insertion of a new string causes branching in a node whose address is already assigned as ID to a string. For example, Fig. 4(a) shows a scenario in which the address of a node (highlighted with bold lines) is associated with string "AB", and then the insertion of string "AC" requires a reorganization of nodes. Since both strings share the prefix "A", our algorithm adds a parent node (highlighted with dashed lines) which holds the prefix, and another node which holds "C" (the remainder of the new string) and the address of this new node is assigned to string "AC". The node which already contained string "AB" is now the child of the new node that contains the prefix "A", therefore, the content of this node is changed to "B" so that still a bottom-up traversal returns back the string "AB".

A second but less frequent case happens when an already inserted string is the prefix of a new string. Figure 4(b) illustrates such a scenario where the address of a node (highlighted with bold lines) is associated with string "AB", but the insertion of string "ABC" requires a new organization of nodes. In this case, the whole string "AB" is a prefix of string "ABC". Therefore, our algorithm adds a new parent node (highlighted with dashed line) which hold the string "AB", and then add the new node "C" (highlighted with dashed line) as its child. The node which previously hosted string "AB" keeps the null string now (highlighted with bold line) so that its memory location is intact, and still a bottom up traversal will reconstruct the original string "AB".

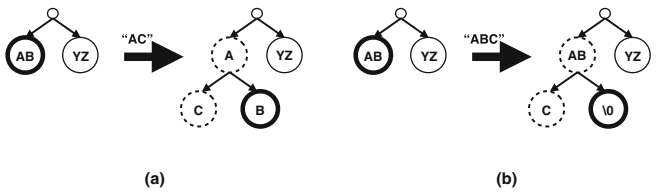


**Fig. 4.** Preserving memory locations already assigned as IDs

[2] This assumption holds in all dictionary encoding implementations we are aware of.

**Differentiating Internal Nodes from Terminal Nodes.** In our implementation, each node contains four fields: *parent*, which is a pointer to the parent node; *string*, which contains the portion of the string that results from path compression and lazy expansion optimizations; *children*, which refers to the first child in the linked list of children, and *sibling*, which links to the next node in the list. Thus, the first child of a node is accessed by first following its *children* link, and the other children are then obtained by following *sibling* links.

The field *children* is used only on the internal nodes. Therefore, we use two different data structures to represent the nodes: one with the field "children" if the node is internal, or without otherwise. This optimization has significant impact on memory efficiency because *any* tree without unary nodes contains more terminal than internal nodes, and this difference is especially more prominent on multiway trees. For instance, on a 64-bit machine, each *children* field occupies at least 8 bytes in memory, and given that the insertion of each unique string will add one leaf to the Trie, for a dataset with 300 M unique terms, we can save 2.2 G of memory only via this optimization.

Notice that we can add this optimization only because our insertion algorithm preserves the constraint on the fixed memory location. Without it, we were unable to change data structure whenever a terminal node becomes an internal one, without breaking the constraint.

## 6 Evaluation

We implemented RDFVault in the Java programming language, and released the source code online[3]. Our implementation stores RDF terms as Java character arrays, therefore it supports an alphabet of 16-bit characters.

As mentioned in Sect. 3, all existing dictionary compression techniques only target static data and thus do not support efficient updates. Hence, we chose an uncompressed conventional dictionary that consists of two hash tables as the baseline (as mentioned in Sect. 3, this is the most popular dictionary implementation that supports fast updates).

For the implementation of this conventional dictionary, we used the GNU Trove Hash Table library (with the default configurations) which is a robust and very memory efficient open-addressing hash implementation. Trove hash allows strings to be directly stored as character arrays which removes the substantial overhead associated with the String objects. Our experiments (not shown here) estimated that the dictionary implemented based on GNU Trove library (hereafter Trove Dictionary) yields more than 20 % better memory efficiency than the implementation based on Java standard HashMap over real-world RDF data.

We do not show a comparison against $B^+$trees because if all data is in main memory then hash tables outperform them significantly. We also did not compare our work against other Trie variants, because we showed in Sect. 4 that they are very memory inefficient for the construction of an RDF dictionary.

[3] https://github.com/bazoohr/RDFVault.git.

To evaluate our approach, we used the four real-world datasets that were considered throughout the previous sections (see Table 1). All experiments were run on a machine equipped with a 32 cores Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60 GHz, and 256 G of memory. The system runs Ubuntu 14.04 and we used Java 1.8.0 with 128 GB of maximum heap space.

To measure the memory consumption, we developed a technique which we were able to validate against existing Java libraries [1]. To measure the performance, we ran each experiment 10 times and report the average value to minimize the overhead of garbage collection on the comparability of our results.

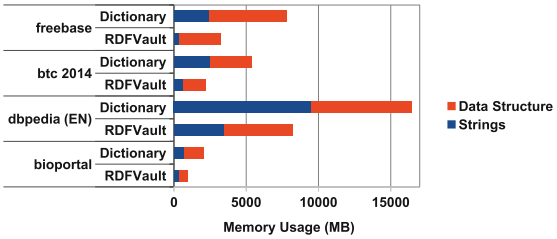


**Fig. 5.** Memory consumption results.

### 6.1 Memory Consumption

*Overall Space Consumption.* Figure 5 shows that RDFVault consumes 50–59% less memory than the conventional dictionary. It is interesting to see that in case of DBPedia (EN), and BTC2014 datasets, the whole memory consumption of RDFVault is less than the memory merely occupied by strings in the conventional dictionary. This clearly shows that RDFVault can successfully exploit the common prefixes of RDF terms to build a highly compact dictionary in memory.

By comparing the results in Fig. 5, with Fig. 1(b), we see that HDT dictionary is considerably smaller than RDFVault, but as we mentioned before this compression is achieved by sacrificing updatability.

*Data Structure Overhead.* Figure 5(a) shows that in all cases the overhead of RDFVault is less than that in conventional dictionary. It also shows that due to the string compression in RDFVault, sometimes strings consume much less space than the data structure (compared to the conventional dictionary). This means that in some cases the data structure becomes the main source of memory consumption in RDFVault. This is the observation that motivated our last optimizations to introduce different data structures to implement the nodes. Further optimization in this direction might be very effective in reducing the overall compression.

Note that although most existing dictionaries consist of two tables, it may also be possible to confine a conventional dictionary into a single hash table. Nonetheless, because strings are shared between the two tables, omitting one

table at best can only reduce the data structure overhead to half, but the memory consumption of strings remains intact. Thus, RDFVault still offers better memory efficiency in all cases (see Figs. 5 and 6).

*IRIs and Literals Compression.* To evaluate the effect of compression on IRIs and literals, we first performed the encoding only on IRIs, and then only on the literals. The results reported in Fig. 6 confirm that eliminating common prefixes is an effective compression technique for IRIs, since it always significantly outperforms the baseline. On the other hand, our technique is less effective in compressing the literals, even though also in this case our method always outperforms the baseline.

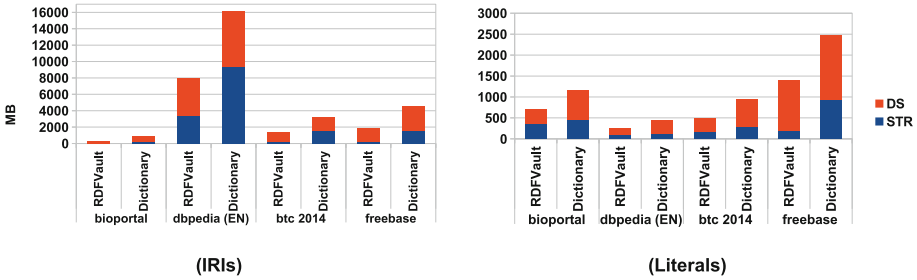


**Fig. 6.** Memory usage considering both IRIs and Literals (DS represents the space occupied by the data structure, while STR is the one taken for strings.

### 6.2 Encoding/Decoding Runtime

We now focus our attention on the impact of our method during encoding and decoding. To this end, we measure the time necessary to encode and decode all terms in the datasets in the order they appear in the publicly available serialization of data (hereafter input order). To be more specific, we also run the same experiment once on IRIs, and once on Literals. Terms were encoded and decoded one after another, and the average encoding/decoding runtime per term is reported in Fig. 7.

As we can see from the left graph, in the worst case it takes about 650ns to encode a term, and about 450ns to decode it. In general, the encoding performance of our approach is comparable to the one of the Trove hash map, and in two cases (BioPortal, BTC 2014) the runtimes are even better. The figure shows that encoding literals is often more expensive both in the conventional dictionary and RDFVault. This suggests that the lower encoding performance of literals could be because they are longer than IRIs. Similar reasons can be given for the slow encoding speed of IRIs in case of DBPedia (EN) dataset, namely because this dataset uses long IRIs, both the conventional dictionary and RDFVault present slower encoding performance than average.

The right graph of Fig. 7 presents the average decoding runtime of RDF terms. The results show that a hash map performs up to 2.5 times better decoding performance, even though in some cases the margin with RDFVault is minimal. In theory the time complexity of both approaches is proportional to the length of string (hash code calculation for conventional dictionary, and string reconstruction in RDFVault), but in practice RDFVault needs to follow multiple references for the bottom up traversal and concatenate the substrings to reconstruct the original one. Therefore, it usually needs to execute more instructions than the conventional dictionary. The positive result is that strings do not have (on average) an excessive length. Therefore, the price that we pay for compressing our input in terms of decoding speed remains rather limited.

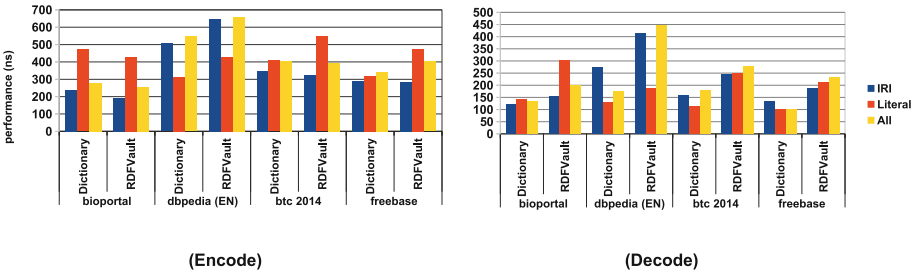


**Fig. 7.** Encoding and decoding runtime of our approach against the baseline.

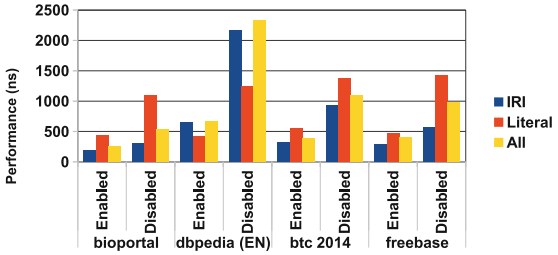


**Fig. 8.** Effect of Move to front policy on encoding performance

### 6.3 Move-to-front Policy Effectiveness

In this experiment, we evaluate the performance gain of applying move-to-front policy on children lists. To that end, we measure the encoding performance of RDFVault over the datasets Table 1 presents with and without move-to-front policy and report the results in Fig. 8.

Results clearly show that the move-to-front policy can effectively achieve up to 3.5 times performance improvement compared to when this optimization

is disabled. It is interesting to see that the effectiveness of the move-to-front policy is not limited to IRIs, and literal terms benefit from this optimization as well.

## 7 Conclusions and Future Work

Dictionary encoding is widely used in semantic web applications, however, recent studies [19] show that sometimes the size of dictionaries becomes even larger than the encoded data. Hence, some efforts (e.g. HDT, and [19]) propose dictionary compression methods that considerably reduce the dictionary size for static datasets. Nonetheless, to the best of our knowledge there is no such technique for dynamic and streaming data. Thus, in this paper, we presented RDFVault, a compact in-memory dictionary that supports dynamic updates.

We first empirically observed a high degree of redundancy in the collection of real-world RDF datasets, and proposed to use Trie data structure to exploit this redundancy in order to compress RDF term in memory. Unfortunately, array-based implementations of this data structure introduce another form of memory inefficiency that stems from excessive number of unused pointers.

We discussed that the only type of Trie that does not suffer from this limitation is a list-based Trie, but unfortunately such Trie is suboptimal compared to array-based ones because of list lookup overhead for generic inputs. To address this last limitation, we introduced a novel list-based Trie that leverages the high degree of similarities and skewness among RDF terms to apply a move-to-front policy in order to reduce the list lookup overhead.

Then, we showed how we use this new list-based Trie as a dictionary named RDFVault, by using as a numerical ID a memory address instead of an integer counter. This design decision removes the need for a dedicated decoding data structure and enhances the memory consumption of the dictionary even further. As a result, we have an in-memory dictionary which both compresses the strings, and supports dynamic updates, and is also confined into a single data structure which reduces the memory consumption.

Our experiments show that the memory consumption of RDFVault is less than half of a conventional dictionary (50–59% less) while it offers comparable (and sometimes even better) encoding speed, though the decoding performance is degraded up to 2.5 times. Given that the decoding in a conventional dictionary is very fast (requires only a single table lookup), we believe that this is still a fairly good performance and a reasonable price to pay for higher compactness level, especially for applications that require frequent encoding or decoding such as stream processing frameworks.

For the future work we intend to extend this study further by exploring the possibilities of multiple concurrent dictionary updates in RDFVault.

**Acknowledgment.** This project was partially funded by the COMMIT project, and by the NWO VENI project 639.021.335.

## References


1. java.sizeof. http://sizeof.sourceforge.net/
2. Askitis, N., Sinha, R.: Hat-trie: a cache-conscious trie-based data structure for strings. In: Proceedings of the Thirtieth Australasian Conference on Computer Science, vol. 62, pp. 97–105. Australian Computer Society Inc (2007)
3. Auer, S., Bizer, C., Kobilarov, G., Lehmann, J., Cyganiak, R., Ives, Z.G.: DBpedia: a nucleus for a web of open data. In: Aberer, K., Choi, K.-S., Noy, N., Allemang, D., Lee, K.-I., Nixon, L.J.B., Golbeck, J., Mika, P., Maynard, D., Mizoguchi, R., Schreiber, G., Cudré-Mauroux, P. (eds.) ASWC 2007 and ISWC 2007. LNCS, vol. 4825, pp. 722–735. Springer, Heidelberg (2007)
4. Binnig, C., Hildenbrand, S., Färber, F.: Dictionary-based order-preserving string compression for main memory column stores. In: SIGMOD. ACM (2009)
5. Cheng, L., Malik, A., Kotoulas, S., Ward, T.E., Theodoropoulos, G.: Efficient parallel dictionary encoding for RDF data
6. Cleary, J.G., Witten, I.: Data compression using adaptive coding and partial string matching. IEEE Trans. Commun. **32**(4), 396–402 (1984)
7. De La Briandais, R.: File searching using variable length keys. In: Papers Presented at the 3–5 March 1959, Western Joint Computer Conference. ACM (1959)
8. Erling, O., Mikhailov, I.: RDF support in the Virtuoso DBMS. In: Networked Knowledge-Networked Media, pp. 7–24. Springer (2009)
9. Fernández, J.D., Gutierrez, C., Martínez-Prieto, M.A.: RDF compression: basic approaches. In: WWW, pp. 1091–1092. ACM (2010)
10. Fernández, J.D., Martínez-Prieto, M.A., Gutierrez, C.: Compact representation of large RDF data sets for publishing and exchange. In: Patel-Schneider, P.F., Pan, Y., Hitzler, P., Mika, P., Zhang, L., Pan, J.Z., Horrocks, I., Glimm, B. (eds.) ISWC 2010, Part I. LNCS, vol. 6496, pp. 193–208. Springer, Heidelberg (2010)
11. Fredkin, E.: Trie memory. Commun. ACM **3**(9), 490–499 (1960)
12. Google. Freebase data dumps. http://download.freebase.com/datadumps
13. Heinz, S., Zobel, J., Williams, H.E.: Burst tries: a fast, efficient data structure for string keys. ACM TOIS **20**(2), 192–223 (2002)
14. Käfer, T., Harth, A.: Billion triples challenge data set (2014). Downloaded from http://km.aifb.kit.edu/projects/btc-2014/
15. Knuth, D.E.: The Art of Computer Programming, Volume 3: Sorting and Searching. International Monetary Fund (1998)
16. Kotoulas, S., Oren, E., Van Harmelen, F.: Mind the data skew: distributed inferencing by speeddating in elastic regions. In: WWW. ACM (2010)
17. Lassila, O., Swick, R.R.: Resource description framework (RDF) model and syntax specification (1999)
18. Leis, V., Kemper, A., Neumann, T.: The adaptive radix tree: ARTful indexing for main-memory databases. In: 2013 IEEE 29th International Conference on ICDE, pp. 38–49. IEEE (2013)
19. Martínez-Prieto, M.A., Fernández, J.D., Cánovas, R.: Querying RDF dictionaries in compressed space. ACM SIGAPP **12**(2), 64–77 (2012)
20. Morrison, D.R.: PATRICIA-practical algorithm to retrieve information coded in alphanumeric. JACM **15**(4), 514–534 (1968)
21. Neumann, T., Weikum, G.: RDF-3X: a RISC-style engine for RDF. VLDB **1**(1), 647–659 (2008)
22. Noy, N.F., Shah, N.H., Whetzel, P.L., Dai, B., Dorf, M., Griffith, N., Jonquet, C., Rubin, D.L., Storey, M.-A., Chute, C.G., et al.: Bioportal: ontologies and integrated data resources at the click of a mouse. Nucleic Acids Res. **37**, W170–W173 (2009)

23. Sussenguth Jr., E.H.: Use of tree structures for processing files. Commun. ACM **6**(5), 272–279 (1963)
24. Urbani, J., Maassen, J., Bal, H.: Massive semantic web data compression with MapReduce. In: HPDC, pp. 795–802. ACM (2010)
25. Yuan, P., Liu, P., Wu, B., Jin, H., Zhang, W., Liu, L.: TripleBit: a fast and compact system for large scale RDF data. VLDB **6**(7), 517–528 (2013)