A Compact RDF Store using Suffix Arrays [*]

Nieves R. Brisaboa[1], Ana Cerdeira-Pena[1], Antonio Fariña[1], and
Gonzalo Navarro[2]

[1] Database Lab., University of A Coruña, Spain.
{brisaboa,acerdeira,fari}@udc.es
[2] Dept. of Computer Science, University of Chile, Chile. gnavarro@dcc.uchile.cl

Abstract. RDF has become a standard format to describe resources in
the Semantic Web and other scenarios. RDF data is composed of triples
($subject, predicate, object$), referring respectively to a resource, a prop-
erty of that resource, and the value of such property. Compact storage
schemes allow fitting larger datasets in main memory for faster process-
ing. On the other hand, supporting efficient SPARQL queries on RDF
datasets requires index data structures to accompany the data, which
hampers compactness. As done for text collections, we introduce a *self-
index* for RDF data, which combines the data and its index in a single
representation that takes less space than the raw triples and efficiently
supports basic SPARQL queries. Our storage format, *RDFCSA*, builds
on compressed suffix arrays. Although there exist more compact repre-
sentations of RDF data, *RDFCSA* uses about half of the space of the raw
data (and replaces it) and displays much more robust and predictable
query times around 1–2 microseconds per retrieved triple. *RDFCSA* is 3
orders of magnitude faster than representations like MonetDB or RDF-
3X, while using the same space as the former and 6 times less space than
the latter. It is also faster than the more compact representations on
most queries, in some cases by 2 orders of magnitude.

1 Introduction

The amount of data publicly available on the Web has been growing steadily over
the years. Many valuable resources are included in this gigantic repository, but
in many cases they are underutilized because of the lack of a common storage
format that allows those resources be automatically identified and accessed. The
Web of Data is an effort to structure the data published by resource providers in
a way that it can be discovered and used under a standard protocol in automatic
form. The Web of Data builds on the principles of the Semantic Web [9].

The *Resource Description Framework* (RDF) [19] provides a simple and pow-
erful way to structure and link data. It uses triples (*subject*, *predicate*, *object*) to

model knowledge, in such a way that a value (*object*) for a property (*predicate*) of a given resource (*subject*) is represented. The adoption of RDF by the W3C as the recommended format to publish information [1] has boosted the growth of RDF repositories and RDF management systems that make up the basis of the current Web of Data. Those systems not only store the RDF data, but they also support queries on it via the SPARQL query language [23].

The increasing interest in the management of RDF repositories (also called RDF stores) is witnessed by the various storage schemes proposed in recent years, which go from those based on relational databases [25] to native solutions such as BITMAT [6], RDF-3X [22], HEXASTORE [26], MonetDB [2], or WaterFowl [12]. As the RDF repositories grow in size, scalability issues challenge the use of RDF storage schemes [18]. A recent work (*K2Triples*) [4] succeeded at reducing both the space usage of previous techniques and their performance to answer basic SPARQL queries: the so-called *basic graph patterns* that make up the primitive SPARQL operations and the algorithms for *merge* and *join*.

In this paper we introduce another storage scheme we call *RDFCSA*. It is based on Sadakane's *Compressed Suffix Array (CSA)* [24], which can represent a text collection in compressed space while supporting pattern searches on it. We modify the *CSA* so as to index a set of triples in a way that all the basic graph patterns of SPARQL boil down to pattern searches on the modified *CSA*. The result is a representation that uses about twice the space of *K2Triples*, but it is faster in most queries, up to 2 orders of magnitude in some cases, which include the most common ones in real-life SPARQL queries [5]. Compared to other representations, *RDFCSA* uses about the same space as MonetDB and 6 times less than RDF-3X, and it is 3 orders of magnitude faster than both.

2   Basic Concepts

2.1   State of the art: *K2Triples*

A RDF dataset can be seen as a set $\mathcal{R}$ of triples (s, p, o) where s, p, and o are respectively a *subject*, a *predicate*, and an *object*. It can also be seen as a connected graph where *subjects* and *objects* are nodes that are connected via arcs labeled by a given *predicate* [19]. Figure 1 shows an example with (not really) fictitious data about the SPIRE conference and some attendants. In the left part, we show the source triples and the underlying RDF graph.

K2Triples [4] tackles the scalability problem of RDF datasets by focusing in reducing their space usage. The authors used two main areas:

(a) Reducing the size of the representation of the strings in the triples through a compressed string dictionary [20, 6, 14]. Each original triple is then represented by a triple of integer ids provided by the dictionary. The right part of Figure 1 depicts the dictionary organization used, and the final set of id-based triples.
(b) Representing the id-based triples in a compact (and indexed) way. The fact that the number of predicates (n_p) in a RDF dataset is typically very small

is exploited by *K2Triples*, which resorts to vertical partitioning [3]: for each *predicate*, it stores the *subjects* that are connected to each *object*. Each such binary relation is generally sparse, so it is represented with a compact k^2-tree data structure [10], which performs well on those relations. The k^2-tree of each predicate can efficiently list the *subjects* related to a given *object* or the *objects* related to a given *subject*.

Simple graph patterns are the most basic SPARQL queries. They are triples where each component can be fully specified as a string (S, P, or O) or left unspecified or "unbounded" ($?S$, $?P$, or $?O$). Such a pattern matches all triples where the specified strings match. For example, in Figure 1, pattern ($?S$, attends, SPIRE) returns the 3rd, 4th and 5th triples listed on "Original RDF Triples".

Due to the vertical partitioning of *K2Triples*, patterns with a fixed *predicate*, that is, (S, P, O), $(?S, P, O)$, $(S, P, ?O)$, and $(?S, P, ?O)$, can be efficiently solved within a unique k^2-tree, whereas patterns with unbounded *predicate* $((S, ?P, O)$, $(?S, ?P, O)$, $(S, ?P, ?O)$, and $(?S, ?P, ?O))$ would involve accessing all the n_p k^2-trees. The *K2Triples* structure partially overcomes this issue by adding two auxiliary indexes, SP and OP, that respectively keep which *subjects* (s) or *objects* (o) occur in a triple related to each *predicate p*. Indexes SP and OP yield large speedups, while typically costing 20%–30% further space.

K2Triples is shown to improve the space of the best state-of-the-art alternatives by a factor of 1.5–12, whereas it matches or outperforms them all in simple graph patterns [4].

2.2   Compressed suffix arrays

Given a string $S[1, n]$ over alphabet $\Sigma = [1, \sigma]$, the *suffix array* $A[1, n]$ is a permutation of $[n]$ so that $S[A[i], n]$ is the ith lexicographically smallest suffix in S. Thus the range of suffixes starting with a search pattern $\alpha[1, m]$ (i.e., the occurrences of α in S) can be binary searched in A in time $O(m \log n)$.

Sadakane's *CSA* [24] represents S and A using two structures (plus others that we ignore in this paper). The first is a bitmap $D[1, n]$, where the 1s mark the first suffixes starting with each distinct symbol in A (i.e., $D[i] = 1$ iff $i = 1$ or $S[A[i]] \neq S[A[i-1]]$). The second *CSA* structure is the array $\Psi[1, n]$, where
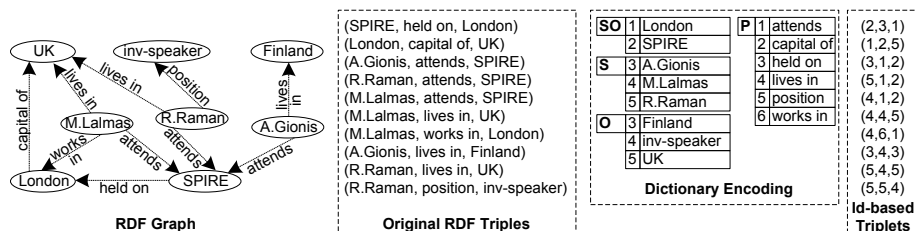

Fig. 1. Example of RDF graph and dictionary encoding in *K2Triples*. **SO** entries in the Dictionary represent terms that act as both *subjects* and *objects* in some triples.

$\Psi[i] = A^{-1}[(A[i] \bmod n) + 1]$. That is, if $A[i] = j$ points to the suffix $S[j, n]$, then $A[\Psi[i]] = j + 1$ points to the next text suffix, $S[j + 1, n]$.

In this paper we assume that every symbol in Σ appears at least once in S. Then $S[A[i]] = rank_1(D, i)$, where $rank_1(D, i)$ is the number of 1s in $D[1, i]$. Moreover, $S[A[i] + 1] = S[A[\Psi[i]]] = rank_1(D, \Psi[i])$, and in general $S[A[i] + j] = rank_1(D, \Psi^j[i])$. Operation $rank$ can be solved in constant time after building an $o(n)$-bit structure on D [11]. Therefore, D and Ψ are sufficient to extract any string $S[A[i], A[i] + m - 1]$ in time $O(m)$. As a result, the binary search on A can be simulated on D and Ψ in the same $O(m \log n)$ time, and the first ℓ symbols of any matching suffix can be extracted in $O(\ell)$ time as well. Array Ψ can be stored in $nH_0(S) + O(n \log H_0(S)) \leq n \log \sigma + O(n \log \log \sigma)$ bits while supporting constant-time access, where $H_0(S)$ is the zero-order empirical entropy of S [24]. Array Ψ is compressible because it is formed by σ increasing subsequences, which can be differentially encoded using δ-codes. By giving special codes to the runs of consecutive 1s in the differences, the space gets closer to higher-order entropies of S [21]. Sampled Ψ values at regular intervals yield fast random access to Ψ.

Our $RDFCSA$ is based on the *integer-based CSA* ($iCSA$)[3] [13], which is a variant Sadakane's CSA that is optimized for large (integer-valued) alphabets. The $iCSA$ reaches the best compression when using truncated Huffman coding of differences and run lengths.

3 $RDFCSA$: A Compressed Suffix Array for RDF

An RDF collection is a set $\mathcal{R}$ of triples (s, p, o) where s, p, and o are respectively a *subject*, a *predicate*, and an *object*. We use the same dictionary encoding as in previous work [4] so that from now on the triple components s, p, and o are regarded as integer ids in the ranges $s \in [1, n_s]$, $p \in [1, n_p]$, and $o \in [1, n_o]$.

3.1 Structure

The first step to build our $RDFCSA$ is to create an ordered list with the n triples from $\mathcal{R}$, and regarding it as a sequence $S_{id}[1, 3n]$ with $3n$ elements. Since the order is not relevant in a set of triples, we sort them by *object*, then by *predicate* and finally by *subject*. We obtain a sequence of integers $S_{id}[1, 3n] = \langle s_1, p_1, o_1, s_2, p_2, o_2, \ldots, s_n, p_n, o_n \rangle$.

To have disjoint subalphabets Σ_s, Σ_p, and Σ_o for the n_s *subjects*, the n_p *predicates*, and the n_o *objects*, we set an array $gaps[0, 2] = [0, n_s, n_s + n_p]$ and convert sequence $S_{id}[1, 3n]$ to $S[1, 3n]$, where $S[i] = S_{id}[i] + gaps[(i-1) \bmod 3]$. Sequence S ranges over alphabet $\Sigma = [1, n_s + n_p + n_o]$, where values $[1, n_s]$ are reserved to *subjects*, $[n_s + 1, n_s + n_p]$ to *predicates*, and the rest to *objects*. We can obviously recover the original triples from S. Then, we build an $iCSA$ on S.

Due to our alphabet mapping, every *subject* is smaller than every *predicate*, and this in turn is smaller than every *object*. Then, the suffix array A of S will have three ranges: $A_s = A[1, n]$, $A_p = A[n + 1, 2n]$ and $A_o = A[2n + 1, 3n]$

---

[3] http://vios.dc.fi.udc.es/indexing/wsi/

where each range points to suffixes starting with a *subject*, a *predicate*, or an *object*, respectively. Array Ψ also has three separate ranges. Entries in $\Psi[1, n]$ will contain values in the range $[n+1, 2n]$ (corresponding to the range of *predicates*). Entries in $\Psi[n + 1, 2n]$ will contain values in the range $[2n + 1, 3n]$ (of *objects*). Finally, entries in $\Psi[2n+1, 3n]$ will contain values in the range $[1, n]$ (of *subjects*).

In a regular *CSA*, if $A[i]$, for $i \in [2n + 1, 3n]$, points to the *object* (third component) of the kth triple of S (i.e., $A[i] = 3k$), then $j = \Psi[i]$ will indicate the position such that $A[j]$ points to the *subject* (first component) of the $(k + 1)$th triple in S (i.e., $A[j] = 3k+1$). This is the key feature that allows traversing the string S virtually using Ψ.

For our purposes, it is more useful that Ψ cycles around the components of the same triple, instead of advancing to the next one. The *RDFCSA* modifies array Ψ so that values in $\Psi[2n + 1, 3n]$ point not to the *subject* of the *next* triple in S, but to the *subject* of the *same* triple. Given the way we have sorted the triples in S, it turns out that $A[i] = 3(i - 1) + 1$, and therefore all we have to do to make Ψ cycle through the same triples is to set $\Psi[i] \leftarrow \Psi[i] - 1$ for all $i \in [2n + 1, 3n]$ (or $\Psi[i] \leftarrow n$ if $\Psi[i] = 1$).

With this modified Ψ we can start at the position $A[i]$ pointing to any place inside a triple (s, p, o) and recover the triple by successive applications of Ψ. For example, if $A[i]$ points to p, then $p = rank_1(D, i)$, $o = rank_1(D, \Psi[i])$, and $s = rank_1(D, \Psi[\Psi[i]])$. If we take Ψ once more we return to $i = \Psi[\Psi[\Psi[i]]]$. In particular, we can retrieve the kth triple of S by starting the process from $A[i]$, which we know points to the *subject* because $i \in [1, n]$. This property will also allow us reduce any simple graph pattern query to the search for a short pattern in S using the *CSA*, and then extract the contents of the resulting triples.

Figure 2 shows the final structure of a *RDFCSA* created over the ten triples included in Figure 1. In this case we have $n = 10$, $n_s = 5$, $n_p = 6$, and $n_o = 5$. The first of the sorted set of source triples is $S_{id}[1, 3] = (1, 2, 5)$, the second is $S_{id}[4, 6]$, and so on. By adding $gaps[0, 2]$ to the triples in S_{id} we obtain $S[1, 30]$. We show the suffix array A built on S and the structures D and Ψ that make up the *RDFCSA* (Ψ is already modified from the original array, Ψ_{orig}, to cycle through each triple). We mark the boundaries of the three ranges $[1, 10], [11, 20]$, and $[21, 30]$. We verify that entries in $A[1, 10]$ point to positions in $S[3k + 1]$, those in $A[11, 20]$ to $S[3k + 2]$, and those in $A[21, 30]$ to $S[3k]$. For example,
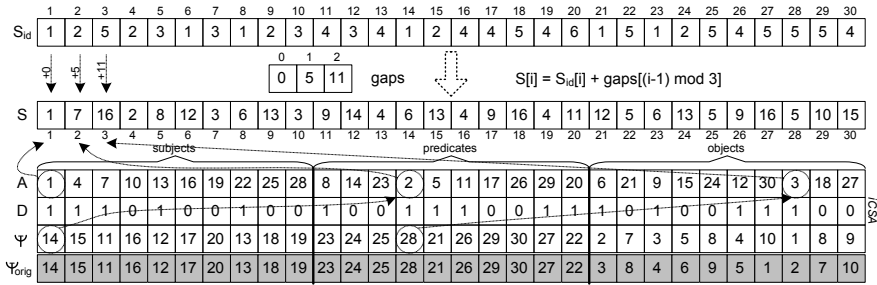

Fig. 2. Structures involved in the creation of a *RDFCSA* for the graph in Figure 1.

$(rank_1(D, 1), rank_1(D, \Psi[1]), rank_1(D, \Psi[\Psi[1]])) = (1, 7, 16)$ recovers the triple in $S[1, 3]$. Also, the third source triple $S_{id}[7, 9]$ can be recovered by doing $S_{id}[7, 9] = (S[7] - gaps[0], S[8] - gaps[1], S[9] - gaps[2])$.

In the $RDFCSA$, the modified array Ψ is represented as in the $iCSA$ [13]. Bitvector D uses a fast $rank$ structure that uses $0.375n$ bits, also as in the $iCSA$. We will also need operation $select_1(D, j)$, which finds the position of the jth 1 in D. It is implemented by a binary search on the $rank$ directories.

We note that enforcing the property $\Psi^3[i] = i$ on our $RDFCSA$ is analogous to the more general *permuterm index* [17]. They index a set of strings as if they were circular, so that patterns of the form $\alpha * \beta$ can be found by searching for the substring $\beta\$\alpha$, where \$ is the string terminator. However, the permuterm index is built on an *FM-index* [15], which on large alphabets like our $[1, n_s + n_p + n_o]$ is implemented on a wavelet tree [16]. This implementation poses a time overhead factor $O(\log(n_s + n_p + n_o))$ for the operation equivalent to computing Ψ, which renders the *FM-index* inferior to the *CSA* on large alphabets [13]. We checked this by using the best *iSSA* variant from [13] to represent sequence S. We tuned *iSSA* to use the same space as $RDFCSA$ (around 60% the size of S regarded as 32-bit integers). Query time to solve (S, P, O) patterns was around $2.5 - 4$ times slower than in $RDFCSA$. Newer alternatives to wavelet trees on large alphabets are only slightly better when implementing FM-indexes [8]. This is why we opt for implementing the technique on top of the $iCSA$ for the case of RDF triples.

3.2   Supporting basic graph pattern queries in $RDFCSA$

Searching for triple patterns is the base to support more complex SPARQL queries on an RDF store. We first show how the 8 primitive operations (S, P, O), $(?S, P, O)$, $(S, ?P, O)$, $(S, P, ?O)$, $(?S, ?P, O)$, $(S, ?P, ?O)$, $(?S, P, ?O)$, $(?S, ?P, ?O)$ can be solved on $RDFCSA$. Then we discuss some RDF-specific optimizations.

The pattern $(?S, ?P, ?O)$ is treated differently because it retrieves all the triples in the dataset (thus it is not really useful as a query). If needed, it can be solved by retrieving every ith triples as described above. The other 7 patterns will be solved by an initial search followed by a traversal to recover the contents of the matching triples.

Binary $iCSA$ search for triple patterns: As explained, the $iCSA$ can run a binary search for the range $A[l, r]$ pointing to the suffixes that start with any pattern $\alpha[1, m]$, so that α appears in S at positions $A[i]$ for $i \in [l, r]$. Then, it can use Ψ to recover the symbols $S[A[i], *]$ for any such i.

In our case, we can solve query (S, P, O) by searching for $\alpha[1, 3] = SPO$, thus determining if it exists in the dataset $(l = r)$ or not $(l > r)$. Further, we can solve queries $(S, P, ?O)$ and $(?S, P, O)$ by searching for $\alpha[1, 2] = SP$ and $\alpha[1, 2] = PO$, respectively. Because Ψ cycles over the triples, we can retrieve the resulting triples in either case, starting from each $i \in [l, r]$. Further, we can also solve queries $(S, ?P, O)$ by searching for $\alpha[1, 2] = OS$, since Ψ regards the triples as circular strings. When only S, P, or O are specified, we must simply search for $\alpha[1, 1] = S$, $\alpha[1, 1] = P$, or $\alpha[1, 1] = O$. We give an example of each case:

- (S, P, O): We set $\alpha[1,3] = [S + gaps[0], P + gaps[1], O + gaps[2]]$, and obtain the range $[l, r]$ with the $iCSA$ binary search. If $l = r$ then (S, P, O) is in the set, otherwise it is not.
- $(S, ?P, O)$: We set $\alpha[1,2] = [O + gaps[2], S + gaps[0]]$, and find the interval $[l, r]$ with the $iCSA$. The number of answers is $r - l + 1$. For each $i \in [l, r]$, we return the triple $(S, rank_1(D, \Psi[\Psi[i]]) - gaps[1], O)$.
- $(?S, P, ?O)$: We set $\alpha[1,1] = [P + gaps[1]]$, and find the interval $[l, r]$ with the $iCSA$ (note that this does not require binary search on Ψ: $l = select_1(D, \alpha[1])$ and $r = select_1(D, \alpha[1] + 1) - 1$). The number of answers is $r - l + 1$ and for each $i \in [l, r]$, the triple $(rank_1(D, \Psi[\Psi[i]]) - gaps[0], P, rank_1(D, \Psi[i]) - gaps[2])$ is recovered.

By using binary search on the $iCSA$, all the triple pattern queries cost $O(r - l + \log n)$, where $r - l + 1$ is the number of occurrences retrieved. In practice, the compression of Ψ introduces important space/time tradeoffs. If the number of triples retrieved is large, the cost of the binary search is negligible. However, it becomes relevant when only one or a few triples are recovered (e.g., no triple recovering is needed for pattern (S, P, O)).

Our first optimization on the original $iCSA$ aims at improving the accesses to Ψ needed to retrieve the triples. Once $[l, r]$ is determined, we always have to compute $\Psi[i]$ and $\Psi[\Psi[i]]$ for all $i \in [l, r]$ (except on the pattern (S, P, O)). We have sped up the access to a range $\Psi[l, r]$ by sequentially decompressing that range of Ψ. Therefore, we only need to access once the sample preceding $\Psi[l]$ and reach position l; all the subsequent values are immediately decoded. This is especially fast if we are inside a run of consecutive values of Ψ. The remaining accesses to Ψ are random and are not be improved.

The other optimizations aim at decreasing the cost of the binary search for $[l, r]$. Two alternatives strategies, *D-select+forward-check* and *D-select+backward-check*, are discussed below.

D-select+forward-check strategy: During the binary search, the comparison between α and $S[A[i], n]$ might be decided with the first integer comparison. Obtaining $S[A[i]] = rank_1(D, i)$ does not require the application of Ψ. At some moment, however, we start having $S[A[i]] = \alpha[1]$ and must compute Ψ to compare $\alpha[2]$ with $rank_1(D, \Psi[i])$. This isolated access to Ψ can be expensive. Instead, we can proceed as follows. Consider the triple pattern (S, P, O). We first find the intervals $R_s = [l_{S+gaps[0]}, r_{S+gaps[0]}]$, $R_p = [l_{P+gaps[1]}, r_{P+gaps[1]}]$, and $R_o = [l_{O+gaps[2]}, r_{O+gaps[2]}]$. These are computed with *select* on D: $l_c = select_1(D, c)$ and $r_c = select_1(D, c + 1) - 1$. Since Ψ is increasing within those intervals, for each i in R_s we can check if $\Psi[i] \in R_p$. The values i that do not pass this check can be discarded. For those that do, we still have to check if $\Psi[\Psi[i]] \in R_o$, in which case we report an occurrence of the searched triple.

Figure 3 (left) illustrates this scenario, where $R_s = [10, 12]$, $R_p = [200, 300]$, and $R_s = [600, 601]$. Neither $\Psi[10]$ nor $\Psi[12]$ map into range $[200, 300]$, only $\Psi[11]$ does. In addition $\Psi[\Psi[11]]$ maps into the range $[600, 601]$ corresponding to *object* 261. Hence, we report an occurrence of the triple $(8, 4, 261)$.
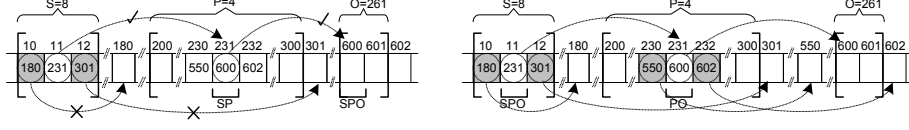
Fig. 3. D-select+forward-check (left) and D-select+backward-search (right) strategies for pattern $(S, P, O) = (8, 4, 261)$.

Computing all the values $\Psi[R_s]$ is much cheaper than computing $|R_s|$ isolated values of Ψ, because of the differential compression of Ψ. In general, there are more objects than subjects, and many more subjects than predicates. Thus, we expect that $|R_o| < |R_s| \ll |R_p|$. If the interval R_s is small enough, this technique may be faster than a standard binary search. Since our Ψ is cyclic, we can start the checking process in interval R_s, R_p, or R_o, so we start from the shortest one.

This procedure is not only applicable to pattern (S, P, O). If we have one unbounded term, we obtain the intervals R_x and R_y corresponding to the bounded terms x and y. Then, we use the same procedure to check whether after applying Ψ to the positions i in the starting interval R_x we fall into R_y or not. For pattern $(?S, P, O)$ we set $x = P$, $y = O$; for pattern $(S, ?P, O)$ we set $x = O$, $y = S$; and for pattern $(S, P, ?O)$ we set $x = S$, $y = P$. Finally, recall that patterns with only one bounded element are directly solved using *select* on D.

D-select+backward-check strategy: Note those i in R_s that pass the check in the previous strategy form a subinterval of R_s, thus we can use binary search to find its limits instead of verifying every $i \in R_s$ one by one. The best way to proceed is known as the backward-search strategy [24]. We show how it can be carried out when searching for pattern (S, P, O). We start in interval $R_o = [l_o, r_o]$, and since Ψ is increasing within interval $R_p = [l_p, r_p]$, we binary search the limits of the subinterval $R_{po} = [l_{po}, r_{po}] \subseteq R_p$ such that $\Psi[i] \in R_o$ for all $i \in R_{po}$. If the subinterval is empty, no match exists. Otherwise, we repeat the same process to find the limits of the subinterval $R_{spo} = [l_{spo}, r_{spo}] \subseteq R_s$ that contain the entries $i \in R_s$ such that $\Psi[i] \in R_{po}$. The final answer is $[l, r] = R_{spo}$.

In Figure 3 (right) we can see that starting in range $R_o = [600, 601]$, when we binary search the interval $\Psi[200, 300]$ for the values that map into range $[600, 601]$, only the entry $\Psi[231]$ remains. Therefore, we obtain the subinterval $R_{po} = [231, 231]$. Now, we binary search the range $\Psi[10, 12]$ for the range that maps to 231 and find that $\Psi[11] = 231$. Then the final interval is $R_{spo} = [11, 11]$.

This strategy is also applicable to patterns $(S, P, ?O)$, $(S, ?P, O)$, $(?S, P, O)$. In the first case we find the subinterval $R_{sp} \subseteq R_s$ that maps via Ψ inside R_p. In the second, the subinterval $R_{os} \subseteq R_o$. In the third, the subinterval $R_{po} \subseteq R_p$.

4 Experimental Evaluation

Our experiments ran on an Amd Phenom-X4-955@3.2GHz CPU, with 8GB DDR2 RAM. The operating system was Ubuntu 12.04 (kernel 3.2.0-31-generic) and the compiler used was gcc 4.6.3 (option -O9). We measure elapsed times.

We evaluated the space/time performance of *RDFCSA* over *Dbpedia*[4], "the nucleus for a Web of Data" [7]. The size of this dataset is around 34GB, containing 232,542,405 triples (2,790,508,860 bytes when regarded as 32-bit integers). The number of different *subjects*, *predicates*, and *objects* is 18,425,128; 39,672; and 65,200,769; respectively. We compared *RDFCSA* with *K2Triples*, MonetDB, and RDF-3X. The recent WaterFowl [12] was not included. Yet, since it reports space 10 times smaller than RDF-3X and similar times [12], we expect it would obtain worse query times than *K2Triples* (see comparison with RDF-3X below) and similar space. Other systems do not run over a dataset of this size [4].

Figure 4 shows the space/time tradeoff of these RDF representations. For *K2Triples* we show two points, corresponding to *K2Triples* and *K2Triples+* [4] (the latter includes the indexes SP and OP that speed up searches with unbounded predicate, see Section 2.1) and we used the tuning recommended by the authors. In the case of *RDFCSA*, the lines connect four points that correspond to sampling Ψ every t_Ψ values: $t_\Psi \in \{16, 32, 64, 256\}$. MonetDB and RDF-3X store the index on disk; we measure the space they use to operate in memory, and run them in warm state, as in previous work [4].

Results clearly show that *K2Triples* (and even *K2Triples+*) use less space than *RDFCSA* (around a half in the case of *K2Triples*). Still, *RDFCSA* uses around half of the space of a raw representation of the triples (and can reproduce them, apart from supporting searches). This is about the same main memory space used by MonetDB, and 6 times less than RDF-3X.

On the other hand, *RDFCSA* obtains much more stable times than *K2Triples*, below 1–2 μsec per occurrence in all cases with a reasonable sampling. *RDFCSA* is in all cases at least 3 orders of magnitude faster than MonetDB and RDF-3X (the only exception is pattern $(?S, P, ?O)$, where RDF-3X is only twice as slow).

K2Triples still obtains the best time for (S, P, O) patterns, as it only needs to accesses the single cell (S, O) of the k²-tree associated to the predicate P, and this is very fast on the k²-tree. Instead, this is the worst case for *RDFCSA*, which must search for a pattern of length 3 and return at most one occurrence.

We can also see that, even though the performance of *K2Triples* is very poor when solving $(S, ?P, O)$ queries, the indexes SP and OP included in *K2Triples+* help solve $(S, ?P, O)$ queries very efficiently. This is because they discard many of the n_p k²-trees that should be accessed otherwise. Only those predicates P that are related to subject S and also with object O must be considered.

On the remaining queries *RDFCSA* is typically faster than *K2Triples* and *K2Triples+*. In particular, for $(S, P, ?O)$, $(S, ?P, ?O)$, and $(?S, ?P, O)$, *RDFCSA* is up to 2 orders of magnitude faster. The first two of these are the most common queries in real-life SPARQL queries, according to an empirical study [5]. In the case of $(?S, P, ?O)$ and $(?S, P, O)$, *RDFCSA* outperforms *K2Triples+* only when the denser samplings are used.

With respect to the optimizations discussed in Section 3, we can see that *D-select+forward-check* (RDFCSA-f in the plots) outperforms *D-select+backward-check* (RDFCSA-b in the plots) in all the triple patterns with one or no un-

---

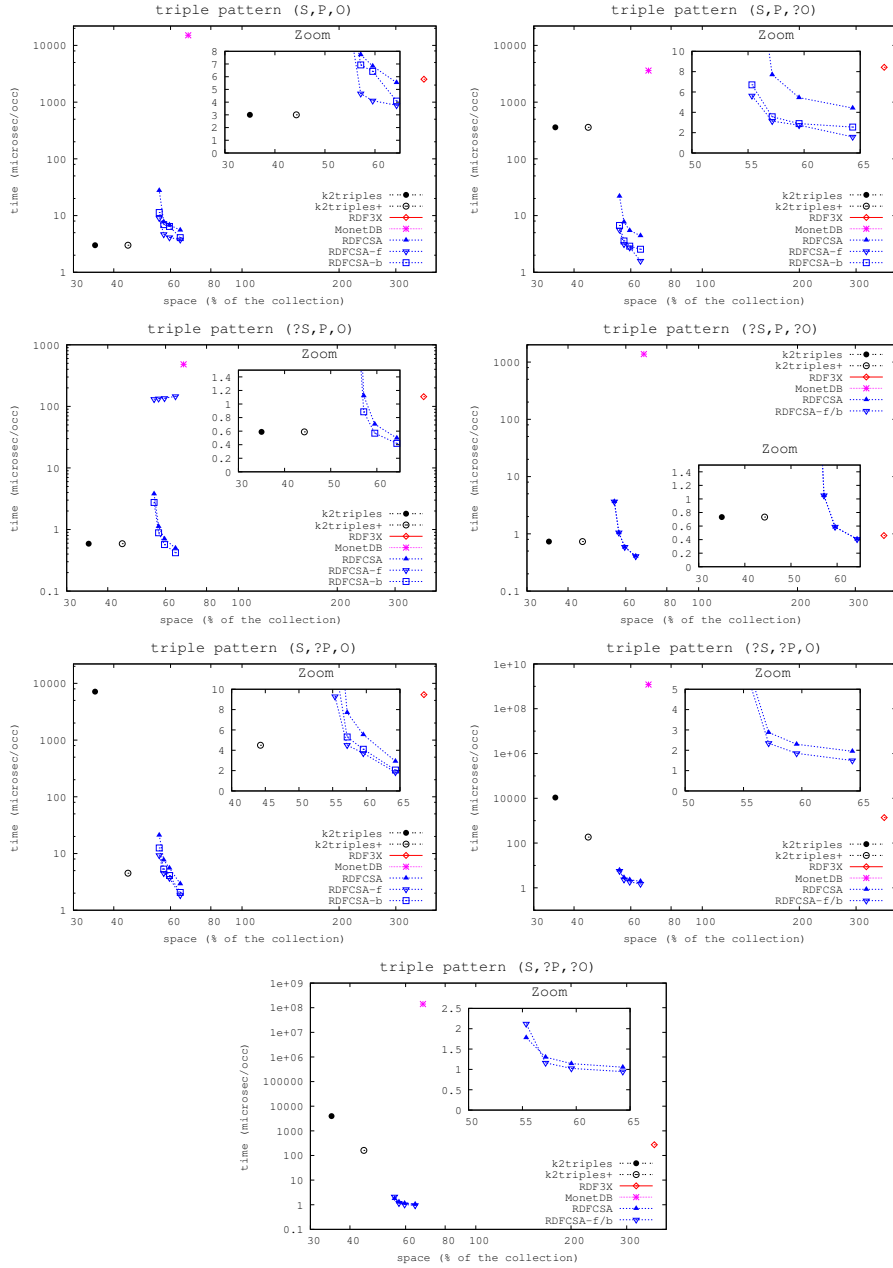[4] http://downloads.dbpedia.org/3.5.1/

Fig. 4. Space/time tradeoff on the primitive SPARQL queries. Space is measured as the percentage of the size of the (in-memory) indexes with respect to the size of the source RDF triples represented as a sequence of 32-bit integers (the dictionary size is not considered here). Query time is the average (in μsec per occurrence) over 500 triple pattern queries of each type obtained from *K2Triples* authors website, `http://dataweb.infor.uva.es/queries-k2triples.tgz`. Note the logscale in the main plots; the zooms use linear scale.

bounded terms (with the exception of $(?S, P, O)$, where it performs very badly because it must sequentially traverse the generally long interval R_p). Except on the pattern (S, P, O), however, the improvement is not significant, and *D-select+backward-check* should be preferred for its more stable and guaranteed performance (of course, we can decide which strategy to use depending on the predicted performance, which can be easily estimated from the sizes of the intervals R_s, R_p, and R_o). In most cases (for example in (S, P, O)), those two strategies reduce the time of the regular *iCSA* binary search to a half or less. In other patterns, such as in $(S, ?P, O)$ those optimizations are the key to match (and even outperform) the performance of *K2Triples+*. Finally, in queries $(?S, ?P, O)$ and $(S, ?P, ?O)$ we can see the slight advantage obtained by performing two *select* operations instead of running the plain binary search with patterns of length 1 (for this experiment, the binary search strategy did not use that speedup).

5   Conclusions and Future Work

We have introduced *RDFCSA*, a competitive structure to self-index RDF data. It builds on an adaptation of the compressed suffix array of Sadakane [24], which is modified to index the RDF triples as cyclic strings, and then various domain-specific optimizations are studied on top of it.

RDFCSA uses about half the space required by the raw data and replaces it. It offers stable and predictable times to solve basic graph patterns (on which more sophisticated SPARQL queries are built), around 1–2 μsec per retrieved triple. Compared to literature standards, *RDFCSA* uses 6 times less space and runs most queries about 1000 times faster than RDF-3X. It uses about the same space as MonetDB, but this is even slower than RDF-3X. There are representations using around half the space of *RDFCSA* [4], their performance is less robust, being up to two orders of magnitude slower than *RDFCSA*, especially on the queries that appear most often in real applications.

Of course, *RDFCSA* handles only basic SPARQL queries, whereas RDF-3X and MonetDB are much more complete. Still, we believe this kernel can be extended to a wider functionality without sacrificing space and taking advantage of its speed when implementing more complex operations. We plan to start by extending *RDFCSA* functionality to handle the join and merge operations of SPARQL. As for the kernel functionality itself, we plan to further study the compressibility of the modified Ψ array (improvements considered in previous work [13] were already tried without success) and faster search algorithms for the particular case of triple patterns.

References

1. RDF 1.1 XML syntax, W3C recommendation. `http://www.w3.org/TR/rdf-syntax-grammar` (2004)
2. MonetDB (2013), `http://www.monetdb.org`
3. Abadi, D.J., Marcus, A., Madden, S.R., Hollenbach, K.: Scalable semantic Web data management using vertical partitioning. In: Proc. VLDB. pp. 411–422 (2007)

4. Álvarez-García, S., Brisaboa, N., Fernández, J., Martínez-Prieto, M., Navarro, G.: Compressed vertical partitioning for efficient RDF management. Knowledge and Information Systems (2014), to appear, preprint at `www.dcc.uchile.cl/gnavarro/ps/kais14.pdf`

5. Arias, M., Fernández, J.D., Martínez-Prieto, M.A., de la Fuente, P.: An empirical study of real-world SPARQL queries. CoRR abs/1103.5043 (2011), `http://arxiv.org/abs/1103.5043`

6. Atre, M., Chaoji, V., Zaki, M.J., Hendler, J.A.: Matrix "bit" loaded: A scalable lightweight join query processor for RDF data. In: Proc. WWW. pp. 41–50 (2010)

7. Auer, S., Bizer, C., Kobilarov, G., Lehmann, J., Cyganiak, R., Ives, Z.: DBpedia: a nucleus for a Web of open data. In: Proc. ISWC/ASWC. pp. 722–735 (2007)

8. Barbay, J., Claude, F., Gagie, T., Navarro, G., Nekrich, Y.: Efficient fully-compressed sequence representations. Algorithmica 69(1), 232–268 (2014)

9. Berners-Lee, T., Hendler, J., Lassila, O.: The semantic Web. Scientific American Magazine (2001)

10. Brisaboa, N., Ladra, S., Navarro, G.: Compact representation of Web graphs with extended functionality. Inf. Syst. 39(1), 152–174 (2014)

11. Clark, D.: Compact PAT Trees. Ph.D. thesis, U. of Waterloo, Canada (1996)

12. Curé, O., Blin, G., Revuz, D., Faye, D.C.: Waterfowl: A compact, self-indexed and inference-enabled immutable rdf store. In: Proc. ESWC'14. pp. 302–316. LNCS 8465 (2014)

13. Fariña, A., Brisaboa, N., Navarro, G., Claude, F., Places, A., Rodríguez, E.: Word-based self-indexes for natural language text. ACM TOIS 30(1), article 1 (2012)

14. Fernández, J.D., Martínez-Prieto, M.A., Gutiérrez, C., Polleres, A., Arias, M.: Binary RDF representation for publication and exchange (HDT). Web Semantics 19, 22–41 (2013)

15. Ferragina, P., Manzini, G.: Indexing compressed texts. J. ACM 52(4), 552–581 (2005)

16. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: Compressed representations of sequences and full-text indexes. ACM Trans. Alg. 3(2), article 20 (2007)

17. Ferragina, P., Venturini, R.: The compressed permuterm index. ACM Trans. Alg. 7(1), article 10 (2010)

18. Jing, Y., Jeong, D., Baik, D.K.: SPARQL graph pattern rewriting for OWL-DL inference queries. Knowl. Inf. Syst. 20(2), 243–262 (2009)

19. Manola, F., Miller, E., (Eds): RDF primer, W3C recommendation. `http://www.w3.org/TR/rdf-primer` (2004)

20. Martínez-Prieto, M.A., Fernández, J.D., Cánovas, R.: Querying RDF dictionaries in compressed space. SIGAPP Appl. Comput. Rev. 12(2), 64–77 (2012)

21. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Comp. Surv. 39(1), article 2 (2007)

22. Neumann, T., Weikum, G.: The RDF-3X engine for scalable management of RDF data. The VLDB J. 19(1), 91–113 (2010)

23. Prud'hommeaux, E., Seaborne, A., (Eds.): SPARQL query language for RDF, W3C recommendation. `http://www.w3.org/TR/rdf-sparql-query` (2008)

24. Sadakane, K.: New text indexing functionalities of the compressed suffix arrays. J. Algorithms 48(2), 294–313 (2003)

25. Sakr, S., Al-Naymat, G.: Relational processing of RDF queries: A survey. SIGMOD Rec. 38(4), 23–28 (2010)

26. Weiss, C., Karras, P., Bernstein, A.: Hexastore: Sextuple indexing for semantic web data management. Proc. VLDB 1(1), 1008–1019 (2008)