Space/time-efficient RDF stores based on circular suffix sorting*

Nieves R. Brisaboa[1,2†], Ana Cerdeira-Pena[1,2†], Guillermo de Bernardo[1,2*†], Antonio Fariña[1,2†] and Gonzalo Navarro[3†]

[1*]Department of Computer Science and Information Technology University of A Coruña, A Coruña, Spain.
[2]CITIC Research Center, A Coruña, Spain.
[3]IMFD, Departament of Computer Science, University of Chile, Santiago, Chile.

*Corresponding author(s). E-mail(s): gdebernardo@udc.es;
Contributing authors: brisaboa@udc.es; ana.cerdeira@udc.es; antonio.farina@udc.es; gnavarro@dcc.uchile.cl;
†These authors contributed equally to this work.

Abstract

The Resource Description Framework (RDF) has gained popularity as a format for the standardized publication and exchange of information in the Web of Data. In this paper we introduce *RDFCSA*, a compressed representation of RDF datasets that in addition supports efficient querying. *RDFCSA* regards the triples of the RDF store as short circular strings and applies suffix sorting on those strings, so that triple-pattern queries reduce to prefix searching on the string set. The RDF store is then represented compactly using a Compressed Suffix Array (CSA), a proved technology in text indexing that efficiently supports prefix searches. Our experiments show that *RDFCSA* is competitive with state-of-the-art alternatives. It compresses the raw data to 60% of its size, close to the most compact alternatives. While most alternatives perform better in some kinds of triple-patterns than in others, RDFCSA features fast and consistent query times, a few microseconds per result in all cases. This enables efficiently supporting join queries by using either merge- or chaining-join strategies over the triple patterns coupled with some

*An early partial version of this article appeared in *SPIRE 2015 [11]*.

specific optimizations such as variable filling. Our experiments on binary joins show that RDFCSA is faster than the alternatives in most cases.

Keywords: Compact data structures, RDF, CSA, Web of Data

1 Introduction

Since the advent of the World Wide Web a few decades ago, the volume of publicly available data has been increasing at a fast pace and has become an invaluable repository of information at global scale, scattered along a large number of repositories from several sources. Since it was originally designed for direct human use, most of such information is stored in the form of unstructured Web pages and hyperlinks between them, which limits our ability to automatically access and process it. The Web of Data, which builds on top of the concepts of the Semantic Web [8], is an effort to provide a formal structure on the data, so that it can be published and processed in automatic form.

The Resource Description Framework (RDF) [27, 40] is a recommendation of the World Wide Web Consortium (W3C) designed to publish and share information in the Web of Data. It is based on a simple labeled-graph-like conceptual structure, but it does not enforce a specific storage format. This graph is usually regarded, for most practical purposes, as a collection of triples, or 3-tuples (source, label, target), that represent the edges in the graph. Going further in the standardization effort, a specific query language called SPARQL has been defined [52] to query RDF collections. SPARQL is based on the concept of triple pattern, a tuple that may contain some unbound elements and that is matched against all the triples in the RDF dataset. Building on this basic selection query, SPARQL enables matching of more complex subgraphs by means of joins, which connect triples that share some component.

The ability of RDF to provide a simple format to publish information has led to its rise in popularity [2]. The lack of an enforced physical representation format has also led to the emergence of many different solutions to efficiently store the RDF data. These solutions, generally called RDF stores or triple stores, aim at providing efficient storage and querying of the RDF dataset. Some RDF stores rely on adapting existing ideas from relational or graph databases [55]. Tools such as Virtuoso [20] and Blazegraph [56], work as fully-functional RDF stores and provide a wide range of query capabilities. Other solutions are based on custom techniques devised specifically for RDF or adapted from other areas. Some examples of these tools include RDF-3X [48], Tentris [9], BITMAT [5], HEXASTORE [57], WaterFowl [18], and HDT [23].

The main issue for modern RDF stores, as the number and size of RDF datasets increases, is the scalability of the solutions [36]. New approaches have been proposed to tackle this problem. Most solutions based on databases or custom indexes rely on caching to maintain good query performance even if the full dataset is too large to fit in main memory. A number of proposals

facilitate scalability by taking advantage of advanced hardware, parallelism or distributed environments [19, 31, 32, 35]. Finally, other solutions aim at achieving very efficient compression so that even large datasets can be efficiently stored and queried in main memory in regular machines, based on compact data structures; K2Triples [3] and permuted trie indexes [51] are examples of proposals that work in this way. Both K2Triples and permuted trie indexes assume that RDF triples are composed of numeric identifiers, so they rely on an external compact dictionary to map RDF strings to identifiers [42, 43].

In this paper we introduce *RDFCSA*, a solution for the compact representation of RDF data that aims at combining good compression with consistently good query performance. *RDFCSA* is based on the compressed suffix array (*CSA*) [54], a data structure originally devised for text indexing that is able to store a set of sequences in compressed space and efficiently supports prefix searches. We modify the *CSA* to regard the triples of the RDF dataset as short circular strings. All the triple-pattern queries can then be transformed into appropriate prefix searches, which are efficiently solved with the *CSA*. Join queries are also implemented by exploiting the query capabilities of the *CSA*. We further engineer the *CSA* to optimize its performance in this scenario.

This work extends our preliminary publication [11]. The main additional contributions of this paper are:

- We provide a detailed description of *RDFCSA* and discuss several improvements made on top of the preliminary work [11]. This includes general optimizations within the internals of the underlying *CSA* structure as well as efficient support of join queries.
- We present three variants of *RDFCSA* that yield different space/time tradeoffs: regular (RDFCSA), best space (RDFCSA-rrr), and best performance (RDFCSA-Hybrid). The regular *RDFCSA* is similar to the one in [11]. Yet, the new optimizations slightly improve space and can halve query times in some cases. *RDFCSA-rrr* reduces space by around 10% at the cost of almost doubling query times. *RDFCSA-Hybrid* is around 2-6 times faster at the cost of typically using 10-20% additional space.
- We include a comprehensive experimental evaluation, comparing *RDFCSA* with state-of-the-art solutions based on compact data structures and other well-known RDF stores. This evaluation expands the preliminary version with the new variants of *RDFCSA*, additional queries (e.g., join queries) and tools (e.g., Virtuoso, Blazegraph).

Our experimental results show that *RDFCSA* provides an excellent space/-time tradeoff, and that thanks to its uniform treatment of all triple patterns, its query times are very consistent and predictable. When compared with other compact solutions, K2Triples obtains better compression but is significantly slower than *RDFCSA*, whereas permuted trie indexes are uniformly faster only when using significantly more space. We have also compared with other popular representations, including HDT, Virtuoso, Blazegraph, MonetDB, RDF-3X, and Tentris; all of these are shown to be far from competitive

with *RDFCSA*, being in most cases several times larger and/or several orders of magnitude slower.

The rest of this paper is organized as follows: Section 2 provides some additional details about RDF, as well as some of the relevant state-of-the-art alternatives, and explains the elements of the *CSA* data structure, necessary to understand our solution. Section 3 describes the *RDFCSA* data structure, and the basic algorithms for simple and advanced queries. Section 4 details the experimental evaluation performed. Finally, Section 5 presents the main conclusions of this work and outlines future work.

2 Previous concepts and related work

2.1 RDF, triple patterns, and SPARQL

The RDF data model is based on a graph-like representation of the data. Given an entity (*subject*), that is associated with a node, each of its properties is represented with an outgoing arc (labeled by a *predicate*) pointing to another node (*object*) that represents the value of that property [40]. An RDF graph can also be seen as a collection of *triples*: an RDF dataset is a set R of triples (s, p, o) (i.e. subject, predicate, object), where each triple represents an arc of the graph. Figure 1 displays an RDF dataset represented with both approaches. Note that we are using simple strings, but in RDF subjects and predicates are identified with Universal Resource Identifiers (URI), and an object may be either a URI or a literal value.

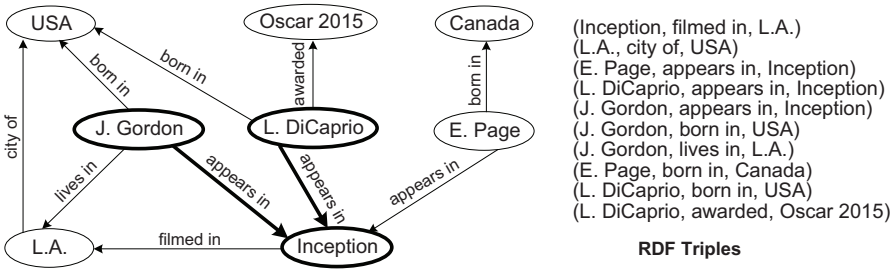

Fig. 1 Example of RDF graph and its representation as a set of triples.

RDF collections can be queried using SPARQL. SPARQL is a complex query language, which builds essentially on triple patterns. A triple pattern is a tuple $(subject, predicate, object)$ where each component may be either *bound* or *unbound*. For instance, the pattern (s, p, o), where all three elements are bound, asks whether subject s has a predicate (or "property") p with value o; the pattern $(s, ?p, ?o)$, where both predicate and object are unbound, asks for all the pairs (p, o) corresponding to the properties of subject s.

SPARQL queries can express more complex conditions using multiple triple patterns, combined using *join variables*, that is, elements of different triple

patterns that must take the same value. For instance, the join operation $(s_1, p_1, ?x) \bowtie (s_2, p_2, ?x)$ (where $?x$ is the join variable) asks for all the objects that are associated to s_1 by property p_1 and to s_2 by property p_2. For instance, to know the movies where both *L. DiCaprio* and *J. Gordon* appeared, we could ask for $(L.\ DiCaprio, appears\ in, ?x) \bowtie (J.\ Gordon, appears\ in, ?x)$, and it would return the movie *Inception*, as highlighted in Figure 1.

Joins can be categorized depending on the position of the join variables. The previous example $(s_1, p_1, ?x) \bowtie (s_2, s_2, ?x)$ is an *object-object* join, because the join variable plays the role of object in both triples; the equivalent *subject-object* and *subject-subject* joins would be $(s_1, p_1, ?x) \bowtie (?x, p_2, o_2)$ and $(?x, p_1, o_1) \bowtie (?x, p_2, o_2)$, respectively. We can also categorize joins according to the unbound elements that appear in each pattern. For instance, $(?s_1, ?p_1, ?x) \bowtie (?s_2, p_2, ?x)$, and $(?s_1, p_1, ?x) \bowtie (?s_2, p_2, ?x)$ are different types of joins because they differ in the number of unbound elements.

A set of triple patterns such as the examples above is usually denoted as a basic graph pattern (BGP). This key component appears in almost all SPARQL queries. Even though a BGP may involve any number of join variables, in this paper we focus on binary-join queries involving just two triple patterns. While the join techniques used in this paper can be easily extended for joins of any number of patterns, the execution order of the joins and the selection of join techniques become more challenging as the BGPs grow.

2.2 RDF stores

As stated before, multiple solutions have been developed to efficiently store and query RDF datasets. The most popular RDF stores are fully functional systems that provide not only storage and query capabilities, but also update mechanisms and integrated SPARQL query endpoints. Virtuoso [20] and Blazegraph [56] are two representative examples of such database solutions.

In addition to those solutions, many other representations have been proposed with varying capabilities and focus, regarding their query support, update abilities, etc. In this paper, we focus on lower-level solutions, and tackle the compact storage of the underlying data by means of compact data structures. We aim at providing fast response times for triple pattern and join queries, without attempting to support all the capabilities of SPARQL and the features of a full database engine. We we introduce next several RDF stores based on different compact data structures that are of special interest to understand our work, as we share some common ideas. A brief summary including the main characteristics of other alternatives can be found in Table 1, in the appendix. A more comprehensive list of these and other state-of-the art solutions for RDF processing can be found in a recent survey [2].

2.2.1 RDF stores based on compact data structures

HDT (from Header-Dictionary-Triples) [22, 23] was originally devised as a serialization format exploiting the usual redundancy in RDF datasets. It gained

popularity [23] thanks to its rather good compression, and its support for basic SPARQL queries [41]. HDT separates the RDF dataset into three main components: *H*eader, *D*ictionary, and *T*riples. The Header component stores metadata, and is not relevant for this paper. The Dictionary stores the different strings appearing in the original RDF dataset, and is in charge of assigning a numeric identifier to each string and providing a bijective string-to-id translation. Finally, the Triples component stores the triples themselves, where each triple is a tuple with three numeric identifiers. This is relevant to our work since *RDFCSA* essentially replaces the Triples component, and is compatible with the dictionary solutions in HDT. HDT defines the decomposition format and provides basic implementations for the dictionary and the triples. Basic solutions for the triples rely on sorted lists that store their elements. HDT can be used to query the data by enhancing the basic structure with additional indexes.

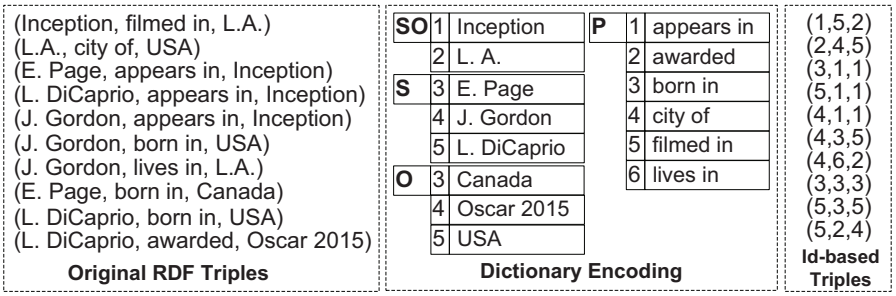

Fig. 2 Dictionary encoding used in HDT for the set of triples in Figure 1.

Figure 2 displays the dictionary encoding used in HDT for the set of triples from Figure 1. Strings are separated in four different sets: set SO contains strings that are both subjects and objects, and three other sets store subjects S, predicates P, and objects O. Each set is sorted in lexicographic order. Entries in SO and P are numbered starting at 1, and entries in S and O are numbered starting at $\mathsf{SO}+1$. This guarantees that each subject, predicate, and object has a unique identifier.

K2Triples [3] is a solution for the compact representation of RDF triples. Like *RDFCSA*, it only considers the structural part of RDF, assuming that triples consist of integer identifiers like those generated by HDT. K2Triples is based on a vertical partitioning [1] of the data. Relying on the fact that the number of predicates (i.e., the number of different properties) is usually very small in RDF datasets, it separates the set of (s, p, o) triples into one set per distinct predicate p, each containing the (s, o) pairs connected by that predicate. Each set of pairs (s, o) is regarded as a binary relation and stored using a k^2-tree [10]. The k^2-tree effectively compresses each binary relation, and its indexing capabilities are exploited to efficiently solve most queries in K2Triples by translating them into basic operations on the k^2-trees.

K2Triples provides specific query algorithms to efficiently answer queries involving joins of two triple patterns. A variant, called K2Triples+, uses two additional indexes SP and OP that drastically improve performance in queries with unbound predicate (the main weakness of vertical partitioning), at the cost of up to 30% extra space. Even with these additional indexes, K2Triples variants are, to the best of our knowledge, the most compact representations of RDF datasets with efficient query support. A more recent representation called **BMatrix** [14] is also based on k^2-trees. However, it is only competitive in some scenarios and lacks support for join operations.

The **permuted trie index** is a recent RDF representation based on the use of compressed tries [51]. The index relies on three permutations (SPO, POS, and OSP) of the triples. Triple-pattern queries are answered by accessing the appropriate structure depending on the fixed variables in the triple pattern. The authors store each permutation as a 3-level trie, and propose several compression techniques based on Partitioned-Elias-Fano (PEF) [50] compression, to obtain very good performance in comparison with other state-of-the-art solutions. In addition to the basic proposal, based on three indexes (which we refer to as *trie-3t*), they also propose solutions that aim at better compression by removing one of the permutations from the index. This saves significant space at the cost of performance loss for queries that used the removed index. Among their variants, the best choice [50, Sec. 4.1] is the one that removes the permutation OSP. We refer to it as *trie-2tp*.

2.3 Rank and select on bitmaps

Bitmaps are fundamental components of many compact data structures. A bitmap $B[1, n]$ is stored in plain form using n bits, but additional structures are required to efficiently support *rank* and *select* operations. The operation $rank_b(B, i)$ counts the number of times bit b appears in $B[1, i]$. The inverse operation, $select_b(B, j)$, finds the position of the jth occurrence of bit b in B. It is possible to compute in constant time both *rank* [17, 34] and *select* operations [17, 44] using $o(n)$ extra bits.

In *RDFCSA*, we only need $rank_1$ and $select_1$ operations, for which we build on a variant that requires $0.375n$ extra bits [28]. It uses a two-level structure that, in the first level (superblocks), stores the cumulative values every 256 positions using $(n/256)$ 32-bit integers, and in the second level (blocks) keeps the cumulative counters relative to the beginning of the corresponding superblock using $(n/32)$ 8-bit integers. We compute $rank_1(B, i)$ by summing the counters at the corresponding superblock and block, and finally scanning a single 32-bit integer to count the number of bits set up to position $i' = (i \quad 1) \mod 32$. This last step is solved with mask-and-shifting, and a small precomputed table to count the number of ones bytewise. This yields $O(1)$ time for $rank_1$. For $select_1$, whose constant-time solution is not so practical, our variant [28] binary searches the values sampled for *rank* in the superblocks, then sequentially scans the counters of the blocks. Therefore, $select_1$ is solved

in $O(\log n)$ time, using the same *rank* structures. We later describe some improvements we make on top of this $select_1$ algorithm.

2.4 Sadakane's Compressed Suffix Array

The *su x array* [39] is a data structure widely used for text indexing. Given a sequence $T[1,n]$, built over an alphabet $\Sigma = [1,\sigma]$, its suffix array is an array $A[1,n]$ that contains a permutation of the integers in $[1,n]$ such that $T[A[i],n] < T[A[i+1],n]$ for all i, in lexicographic order. The suffix array is built by sorting all the suffixes $T[j,n]$ and storing in $A[i]$ the offset in the sequence T of the ith suffix in lexicographical order. Note that all the suffixes starting with the same string α are contiguous in A, and that any occurrence of α in T is the prefix of a suffix of T starting with α. We can then efficiently search for all the occurrences of a pattern $\alpha[1,m]$ in T by two binary searches on its suffix array A, requiring time $O(m \log n)$, which locate the range $A[l,r]$ corresponding to all the positions where α occurs in T.

The original suffix array is useful for searching but requires a significant amount of space, $n \log n$ bits, in addition to the original sequence. Sadakane's Compressed Suffix Array, or *CSA* [54], provides a compact representation that uses at most $n \log \sigma + O(n \log \log \sigma)$ bits and replaces both T and A, while still efficiently supporting searches.

The *CSA* is composed of several data structures. The most important of them is a new permutation $\Psi[1,n]$ [29]. For any i in $[1,n]$, assuming $A[i] = p$, $\Psi[i]$ stores the position j in the suffix array that points to the next position in the original sequence (i.e., $A[j] = A[i]+1 = p+1$). A special case arises when $A[i] = n$, where $\Psi[i]$ is set to j such that $A[j] = 1$. Concisely, Ψ is defined as $\Psi[i] = A^{-1}[(A[i] \bmod n) + 1]$.

In addition to Ψ, a bitmap $D[1,n]$ contains a 1 at the positions in A where the first symbol of the corresponding suffixes changes (i.e., $D[i] = 1$ iff $i = 1$ or $T[A[i]] \neq T[A[i-1]]$). In order to know the symbol in T pointed by $A[i]$, we can count the number of 1s in D up to position i, that is, $rank_1(D,i)$.

Using Ψ and D we can reproduce the same binary search of the suffix array, without storing T or A. The first symbol of the suffix pointed by $A[i]$ can be computed as $rank_1(D,i)$. To extract the following symbols, we iterate using Ψ: $\Psi[i]$ stores the position i' in A that points to the next symbol of the text; therefore, we can extract subsequent symbols as $rank_1(D,\Psi[i])$, $rank_1(D,\Psi[\Psi[i]])$, and so on. Assuming that *rank* operations in D and accesses to Ψ can be computed in constant time, a binary search in the *CSA* still requires $O(m \log n)$ time. After computing the range $A[l,r]$ of the occurrences of α, a forward text context for each can be extracted by iterating with Ψ in the same way.

An uncompressed Ψ array would still require the same space as A. However, Ψ can be partitioned into at most σ increasing contiguous subsequences, which makes it highly compressible by encoding it differentially, i.e. by representing each $\Psi[i]$ as $\Psi[i] - \Psi[i-1]$. A run of t increasing values in $[1,n]$ can be represented in $t \log_2(n/t) + O(t \log \log(n/t))$ using δ-codes. Overall, Ψ can be compressed to space proportional to the zero-order empirical entropy of

the original sequence, or $nH_0(T) + O(n \log H_0(T))$ $n \log \sigma + O(n \log \log \sigma)$ bits [54]. Further improvements, combining the δ-codes with run-length encoding (RLE) for runs of consecutive differences equal to 1 (which tend to appear in Ψ), reduced this space even more and achieved compression proportional to the higher-order entropy of T, $nH_k(T)$ [46].

The *RDFCSA* is based on the *integer-based CSA* (*iCSA*)[1] [21]. The *iCSA* is a variant optimized for large (integer-based) alphabets, with some differences in implementation and compression techniques with the original *CSA*. Particularly, in the *iCSA* the best compression is achieved by using differential encoding of the consecutive Ψ values, followed by mixing Huffman and run-length encoding of the resulting gaps. To provide efficient access (in time $O(t_\Psi)$) to Ψ, absolute Ψ values are stored at positions $\Psi[1 + k \quad t_\Psi], k \quad 0$.

Note that both the *CSA* and the *iCSA* include additional structures to support other text search functionalities. Particularly, they add samplings of A and A^{-1}, to be able to find the position in T of the occurrences of α, or to extract arbitrary substrings. These additional data structures are not necessary in our *RDFCSA*.

3 Our proposal: *RDFCSA*

The two compact approaches we reviewed in the previous section have issues to support all the possible combinations of triple patterns. K2Triples and K2Triples+ are weaker when the predicate is unbound, whereas the permuted trie index favors the triple patterns where there is a trie starting with the bound elements. The key idea of *RDFCSA* is that, if we regard the triples (s, p, o) as circular strings (i.e., the s follows the o again), then for every possible triple pattern there is a rotation of (s, p, o) where all the bound values precede all the unbound ones. Thus, if we index the triples as circular strings, every possible triple pattern can be reduced to a search for the circular strings that start with some prefix. We use the *CSA* to simulate a set of circular strings corresponding to all the triples of the RDF dataset. This approach yields a uniform search approach that will translate into not only fast, but also consistent and predictable, query times.

We follow the convention of treating an RDF dataset as a set R of triples (s, p, o), where s, p, and o are a subject, a predicate, and an object, respectively. Our solution works with integer identifiers (ids) for each of them, so it requires a separate dictionary to perform the translation between the original string values and the corresponding integer ids. Particularly, we base our solution on the same dictionary encoding proposed by HDT and also used by K2Triples, which was described in Section 2. Therefore, we assume a dictionary encoding in which subjects, predicates, and objects are integers in contiguous ranges: $s \; 2 \; [1, n_S]$, $p \; 2 \; [1, n_P]$, and $o \; 2 \; [1, n_O]$ (note the overlapped identifiers in Figure 2). While any other dictionary encoding scheme could be used for our

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

purposes without affecting our implementation, we do take advantage of this particular encoding to perform some optimizations in join queries.

Our *RDFCSA* representation is a *self-index*, meaning that we can recover the triples from it, and thus it replaces the RDF store. As explained, it organizes the triples in a way that can be represented with a modified *CSA* data structure that efficiently answers relevant queries in the domain. We first describe how the data structure is built from the set of triples, and then how we efficiently support the relevant query operations over our self-indexed representation of the triples.

3.1 Data structure

Given an input set R of n triples, we sort them increasingly by subject, then break ties using the predicate and further break ties using the object, to make up a sequence $T_{sort}[1, n]$ of triples. Then, we transform this sequence of tuples into an integer sequence of identifiers $T_{id}[1, 3n]$, by placing the ids of the three components of each entry $T_{sort}[i]$ at consecutive positions $T_{id}[1 + 3(i \quad 1)]$, $T_{id}[2+3(i \quad 1)]$, and $T_{id}[3+3(i \quad 1)]$. Hence, at the end of this step, $T_{id}[1, 3n] = \langle s_1, p_1, o_1, s_2, p_2, o_2, \ldots, s_n, p_n, o_n \rangle$ stores all the ids for the sorted triples.

Next, we transform the identifiers in order to obtain disjoint integer alphabets Σ_s, Σ_p, and Σ_o for the n_s subjects, the n_p predicates, and the n_o objects. This can be performed just by computing the displacements necessary for predicates and objects: we set an array $gaps[0, 2] = [0, n_s, n_s + n_p]$ and convert sequence $T_{id}[1, 3n]$ into $T[1, 3n]$, where $T[i] = T_{id}[i]+gaps[(i \quad 1) \bmod 3]$. After this transformation, our sequence $T[1, 3n]$ has an alphabet $\Sigma = [1, n_s+n_p+n_o]$, where values in the range $[1, n_s]$ are reserved to subjects, those in the range $[n_s + 1, n_s + n_p]$ to predicates, and the remaining ones to objects.

After the previous transformations, which can be trivially reversed to obtain the original set R of triples, we build an *iCSA* on T. However, some key changes have to be performed over the underlying suffix array in order to efficiently answer queries. Those changes rely on specific properties of our construction method.

In particular, we take advantage of the following property of the generated suffix array A: it contains three well-delimited sections $A_s = A[1, n]$, $A_p = A[n + 1, 2n]$ and $A_o = A[2n + 1, 3n]$, corresponding respectively to subjects, predicates, and objects. This is a direct consequence of our construction method, which generates integer identifiers such that every subject is smaller than every predicate, and this in turn is smaller than every object. This ordering means that, when sorting suffixes, entries corresponding to subjects, predicates, and objects end up clustered in different sections. Therefore, A_s contains entries pointing to subjects in T, A_p points to predicates, and A_o points to objects. Accordingly, array Ψ also contains three separate ranges with special properties. Recall that $\Psi[i]$ contains, for the position p such that $A[i] = p$, the position in A that points to the next element $p + 1$ in T. Due to the division of A into three sections, entries in Ψ also point to those delimited intervals, so each region of Ψ contains values in a different range: values of

$\Psi[1,n]$ are in the range $[n+1,2n]$ (pointing to the range of predicates); entries in $\Psi[n+1,2n]$ are in the range $[2n+1,3n]$ (pointing to objects); and entries in $\Psi[2n+1,3n]$ are in the range $[1,n]$ (pointing to subjects).

Since our sequence T contains all the concatenated triples in SPO order, the symbol following an object will always be the subject of the next triple. Therefore, if we are at position i in the suffix array, such that $A[i]$ points to an object (i.e., $A[i]$ for i 2 $[2n+1,3n]$, or $A[i] = 3k$ for some k), when we iterate using Ψ we reach a position j such that $A[j]$ points to the subject of the next triple. The original organization of Ψ was useful in the *CSA* to allow full extraction of the text. In our case, however, we only need to extract individual triples and, further, regard them as circular. Thus, we make Ψ cycle around the components of the same triple, instead of advancing to the next one. Our *RDFCSA* then uses a modified array Ψ in which values within $\Psi[2n+1,3n]$ point not to the subject of the *next* triple in T, but to the subject of the *same* triple. Thanks to the way we ordered the triples before building T, and the grouping of subjects in A, we can compute the modified Ψ very efficiently from the original array: we simply set $\Psi[i]$ $\Psi[i]$ 1 for all positions corresponding to objects (i 2 $[2n+1,3n]$), or $\Psi[i]$ n for the special case $\Psi[i] = 1$.

The modified Ψ provides a simpler way to recover and search triples. Since Ψ cycles over the triples, we can start at any position in the suffix array $A[i]$, and apply Ψ to recover the remaining components of the triple. For instance, if $A[i]$ points to a predicate (i 2 $[n+1,2n]$), we can find the object with an iteration using Ψ, and the subject with a second iteration ($p = rank_1(D,i)$, $o = rank_1(D,\Psi[i])$, $s = rank_1(D,\Psi[\Psi[i]])$). Using the original Ψ we would not be able to iterate from objects to subjects. Note also that only two iterations are necessary for any triple, and if we apply Ψ a third time we return to $i = \Psi[\Psi[\Psi[i]]]$. The same property allows us to reduce any triple pattern to a search for a short string in T. We will further discuss this when describing the query operations for *RDFCSA*.

We note that the modified Ψ used in *RDFCSA*, enforcing the property $\Psi[\Psi[\Psi[i]]] = i$, is similar to the *permuterm index* [25], which tackles a more general case. They also index a set of strings as if they were circular, so that queries involving patterns of the form α β (where stands for an arbitrary string) can be answered by transforming it to the string pattern $\beta\$\alpha$, where \$ is a special string terminator symbol. However, the permuterm index is built on top of an *FM-index* [24], which uses a wavelet tree [26] as the underlying data structure. The wavelet tree implementation requires time logarithmic in the alphabet size, $O(\log(n_s + n_p + n_o))$ in our case, for each basic traversal step, equivalent to a computation of Ψ in our solution. This overhead renders the *FM-index* inferior to the *CSA* on large alphabets [21]. We checked this by comparing the best-performing such variant on integer alphabets [21] to index our sequence T, and obtained times to answer (s,p,o) patterns around 2.5–4 times slower than those in *RDFCSA*. More recent implementations of wavelet trees on large alphabets have shown only minor improvements for FM-indexes

[7]. This is why we implemented our technique on top of the *iCSA* for the case of RDF triples.

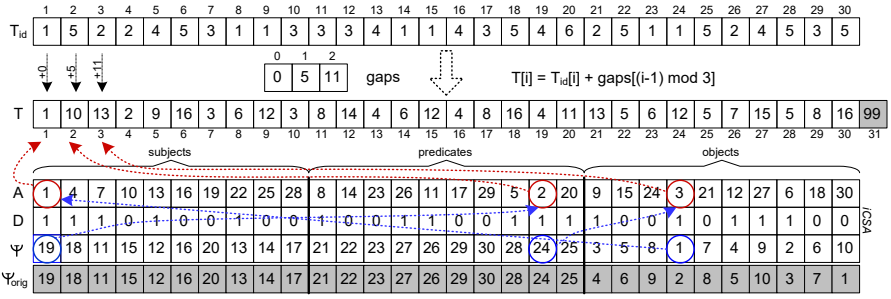

Fig. 3 Structures involved in the creation of a *RDFCSA* for the triples in Figures 1 and 2.

Figure 3 displays the different data structures involved in the creation of a *RDFCSA* for a given set of triples. We use the same triples described in Figure 1, following the dictionary encoding of Figure 2. The collection contains $n = 10$ triples, with $n_s = 5$ subjects, $n_p = 6$ predicates, and $n_o = 5$ objects. The first step is sorting the triples in SPO order, and concatenating their components in array T_{id}: the first triple is located in $T_{id}[1,3] = (1,5,2)$, the second one in $T_{id}[4,6] = (2,4,5)$, and so on until the last triple, which is set in $T_{id}[28,30] = (5,3,5)$. We compute $gaps[0] = 0$, $gaps[1] = n_s = 5$, $gaps[2] = n_s + n_p = 11$, and then create T by adding the appropriate component of $gaps[0,2]$ to the values in T_{id}. At the end of this step we obtain $T[1,30]$. Note that we add an extra entry at the end of T as an implementation trick: by adding this value, larger than any entry in T, we ensure that suffix sorting works properly when constructing the suffix array A, without having to change the construction used by the original *iCSA* (similar results could be obtained by adjusting the algorithm used for suffix comparison). The suffix array A is then built on top of $T[1,30]$ (recall that the last element is added to T just for sorting purposes, but it is not considered as a part of the array itself). Our construction process continues by building the bitmap D and the array Ψ_{orig} as in the original *iCSA*. Then, the final array Ψ used by the *RDFCSA* is created from Ψ_{orig} by subtracting 1 to $\Psi_{orig}[i]$, for each position i in the interval $[21,30]$ corresponding to objects, and finally setting $\Psi[30] = 10$ for the special case where $\Psi_{orig}[30] = 1$ (as indicated above).

The main properties stated for A and Ψ can be easily checked in the example. For instance, entries in $\Psi[1,10]$ contain values in the interval $[11,20]$, entries in $\Psi[11,20]$ contain values within $[21,30]$ and entries in $\Psi[21,30]$ contain values within $[1,10]$. The figure also displays the general procedure to traverse the sequence to recover the first triple: starting at $i = 1$, which corresponds to the subject of the triple, we compute $\Psi[1] = 19$ to locate the predicate, and then compute $\Psi[19] = 24$ to locate its object. Note that if we apply Ψ once again, $\Psi[24] = 1$ takes us back to the subject location due

to the cyclical Ψ. When performing binary search or extracting the triple, we can get the corresponding values by computing $s = rank_1(D, 1) = \mathbf{1}$, $p = rank_1(D, 19)$ $gaps[1] = 10$ $5 = \mathbf{5}$ and $o = rank_1(D, 24)$ $gaps[2] = 13$ $11 = \mathbf{2}$ to recover the original triple $(1, 5, 2)$.

3.1.1 Data structure optimizations

The basic implementation described uses the same data structures as the *iCSA* [21] to store Ψ and D. Precisely, D uses the described structures to support *rank* and *select*, whereas Ψ uses differential encoding combined with Huffman and run-length encoding, which performed best.

On this basic structure, we apply a couple of simple improvements that are specific of the kind of data we are representing. Basically, since the suffix array is separated into three areas of size n, for subjects, predicates, and objects, and these have different characteristics, it pays off to separate D and Ψ into three arrays of length n each: $D_s[1, n]$, $D_p[1, n]$, and $D_o[1, n]$, and $\Psi_s[1, n]$, $\Psi_p[1, n]$, and $\Psi_o[1, n]$. We can then encode each array in different form.

In most RDF datasets, the number P of different predicates is very small. Since D_p has only P 1s, we can avoid the computation of $select_1(D_p,)$ by directly storing a small array of P entries with the results of the P distinct $select_1(D_p,)$ queries; the $select_1$ operations on D_s and D_o are still carried out as described. The effect in the overall space is negligible.

Further, we add a small structure to speed up $select_1$ queries on D_s and D_o: being n^0 n the number of 1s in D , we add an array ($sOnes$) of $n^0/256$ entries where we store the position where every 256th 1 appears in the bitmap. Given a query $select_1(D\ , i)$, the answer can be either stored in our array (if i is a multiple of 256), or it can be between the samples $\lfloor i/256 \rfloor$ and $\lfloor i/256 \rfloor + 1$. We then start the binary search on the range of the corresponding superblocks, which saves in practice most of the binary search cost. The total space for $rank_1$ and $select_1$ queries is $0.5n$ bits for each of D_s and D_o.

The values in Ψ_s, which are in $[n + 1, 2n]$, are decreased by n so that they point inside Ψ_p, and those of Ψ_p, which are in $[2n + 1, 3n]$, are decreased by $2n$, so that they point inside Ψ_o. These reductions do not affect the differential encodings, but they yield a slight gain of space in the absolute samples, which require $\lceil \log_2 n \rceil$ instead of $\lceil \log_2 3n \rceil$ bits.

More importantly, we can represent each partition of Ψ in different form. We define a variant of our data structure that we call *Hybrid*, which slightly increases the space to obtain better access time to Ψ. Concretely, *Hybrid* stores Ψ_s and Ψ_o in plain form, and keeps Ψ_p differentially compressed as described. For Ψ_s and Ψ_o, we use a simple array requiring $\lceil \log_2 n \rceil$ bits per entry. Keeping Ψ_s and Ψ_o uncompressed means that accessing Ψ will be much faster, in time $O(1)$ instead of $O(t_\Psi)$, in these regions. This will be most noticeable on queries that only use those ranges of Ψ.

Choosing a plain representation for Ψ_s and Ψ_o is reasonable because of the characteristics of the *iCSA* and RDF datasets: the numbers S and O of different subjects and objects are relatively large, and therefore we take little advantage

of the fact that $\Psi_S[1,n]$ and $\Psi_O[1,n]$ are formed by S and O increasing runs, respectively: this leads to using $\log_2 S + O(\log\log S)$ or $\log_2 O + O(\log\log O)$ bits to encode each difference, instead of $\log_2 n$ bits to encode an absolute value. For example, using $t_\Psi = 32$, the differential encoding of Ψ_S reduces its size to 93% of the plain size using $\lceil \log_2 n \rceil$ bits, and that of Ψ_O reduces it to around 75%. Instead, because there are few predicates, the differential encoding reduces Ψ_P to around 15% of its uncompressed size. This scheme could be easily generalized so as to apply compression only if a given space reduction is achieved.

For simplicity, we will keep speaking of D and Ψ, ignoring the implementation detail that they are stored in partitioned form.

3.2 Query operations

In this section we describe how to use *RDFCSA* to answer triple-pattern queries, which constitute the main building block to support SPARQL queries. We describe how to solve the 7 triple-pattern queries (s,p,o), $(?s,p,o)$, $(s,?p,o)$, $(s,p,?o)$, $(?s,?p,o)$, $(s,?p,?o)$, $(?s,p,?o)$. The basic operatory for all of these patterns is to locate the range of entries corresponding to their bound components, and then extracting the corresponding triples. We will also describe various RDF-specific optimizations.

We disregard the triple pattern $(?s,?p,?o)$, because it retrieves all the triples in the dataset and is not really useful as a query. Nevertheless, we note that it can be easily solved by omitting the search phase and simply extracting the full set of triples using Ψ.

3.2.1 Solving triple patterns using the regular binary search on the *iCSA*

The *iCSA* can locate all the occurrences of a pattern, by binary searching the range $A[l,r]$ of the suffixes that start with the given pattern. Given a query pattern $\alpha[1,m]$, the range of positions $[l,r]$ in the suffix array A will contain pointers to all the positions in the text where the pattern α occurs. After computing $[l,r]$, Ψ is used to recover the corresponding symbols.

In our case, we are interested in answering a triple-pattern query, where some components can be bound and others unbound. As discussed previously, our modified Ψ allows us to treat all cases similarly, by searching for a subsequence corresponding to the fixed components in the triple pattern. For instance, to answer an (s,p,o) query we build a sequence $\alpha[1,3] = spo$, and use that as our pattern for the binary search in the *iCSA*. To answer $(s,p,?o)$ and $(?s,p,o)$ queries, we search for $\alpha[1,2] = sp$ or $\alpha[1,2] = po$, respectively. We can also answer $(s,?p,o)$ queries by searching for $\alpha[1,2] = os$, thanks to the cyclical traversal of our modified Ψ. Similarly, for query patterns where only one of the elements is fixed, we simply search for $\alpha[1,1] = s$, $\alpha[1,1] = p$, or $\alpha[1,1] = o$. Next we detail the solution for each group of triple patterns, depending on the number of unbound variables.

For (s, p, o) queries, we actually set $\alpha[1,3] = [s + gaps[0], p + gaps[1], o + gaps[2]]$, containing all the elements of the triple pattern. We then perform a binary search for α in the *iCSA*. If $l = r$ then (s, p, o) is an existing triple, otherwise it is not in the dataset.

For queries with a single unbound variable, we proceed similarly with a binary search. Yet, we now have to recover the original triples afterwards. For instance, for $(s, p, ?o)$ queries we set $\alpha[1,2] = [s + gaps[0], p + gaps[1]]$. Binary searching for α in the *iCSA*, we find the interval $[l, r]$ corresponding to the result set. The number of answers is $r \quad l + 1$. For each i 2 $[l, r]$, we return the triple $(s, p, rank_1(D, \Psi[\Psi[i]]) \quad gaps[2])$. Similarly, for $(s, ?p, o)$, we set $\alpha[1,2] = [o + gaps[2], s + gaps[0]]$, then we binary search for pattern α, and return all triples $(s, rank_1(D, \Psi[\Psi[i]]) \quad gaps[1], o)$. For $(?s, p, o)$, we set $\alpha[1,2] = [p+gaps[1], o+gaps[2]]$, we binary search for α, and return the triples $(rank_1(D, \Psi[\Psi[i]]) \quad gaps[0], p, o)$.

For queries with two unbound variables, we can still perform a binary search to locate the occurrences of the bound variable. For instance, for $(?s, p, ?o)$ triple patterns we set $\alpha[1,1] = [p + gaps[1]]$, and find the interval $[l, r]$ with the *iCSA*. The number of results is again $r \quad l + 1$, and for each i 2 $[l, r]$, the triple $(rank_1(D, \Psi[\Psi[i]]) \quad gaps[0], p, rank_1(D, \Psi[i]) \quad gaps[2])$ is recovered. Note that, in this case, the binary search in the *iCSA* does not require a binary search operation on Ψ, since we can compute $l = select_1(D, \alpha[1])$ and $r = select_1(D, \alpha[1] + 1) \quad 1$. As in the previous examples, $(?s, ?p, o)$ and $(s, ?p, ?o)$ can be answered using exactly the same operation but adjusting α and the computation to return the result triples.

Since we are using a binary search on the *iCSA*, all the triple-pattern queries require $O(r \quad l + \log n)$ time, where $r \quad l + 1$ is the number of query results. In addition to this, for most query patterns we need to perform a number of accesses to Ψ per query result in order to return the complete triples. In practice, efficient access to Ψ must be balanced with efficient compression; the compression of Ψ introduces a significant space/time tradeoff that can be tuned in our representation. Note that the space/time tradeoff also depends on the type of query pattern involved: if a query returns a large number of results, the cost of the binary search becomes negligible and the time required to perform accesses to Ψ dominates the cost of the query. However, the binary search cost becomes relevant when only one or a few triples are returned, as well as in (s, p, o) queries, where no triple-pattern retrieval is necessary.

3.2.2 Query optimizations

We now describe a number of optimizations and algorithmic variants that improve our performance.

One enhancement improves query patterns with two unbound terms, in which we always need to perform two *select* operations on D over two consecutive values, i and $i + 1$. Once we compute $j = select_1(D, i)$, we can replace $select_1(D, i + 1)$ by a new operation $selectnext(D, j)$, which finds the next 1 after $D[j]$. We implement *selectnext* by scanning D bytewise from position

$j+1$ to the end of its block. If we find no 1 up to then, we scan the following 32-bit words looking for a nonzero block. If we find no 1 up to then, we check if the next superblock has a 1, and if not, we binary search for the next one that has. On that superblock, which contains the answer, we restart the word-wise scan, then the bytewise scan, and finally use the same table of *select* to find the desired 1. This is in practice faster than a second binary search.

Our next optimization improves the performance of accesses to Ψ, particularly taking into account that in most cases we need to compute values of Ψ for a relatively large range of consecutive positions. In the original algorithm, once $[l, r]$ is determined through binary search, we have to compute $\Psi[i]$ and $\Psi[\Psi[i]]$ for all i 2 $[l, r]$ to retrieve the missing elements in each triple (except on the pattern (s, p, o)). Since Ψ is differentially encoded, each access takes time $O(t_\Psi)$, where we spend $(n \log n)/t_\Psi$ bits to store the absolute samples. In order to improve the speed of these accesses, we sequentially decompress the whole range $\Psi[l, r]$. This means that, once we decode $\Psi[l]$ in $O(t_\Psi)$ time, all the subsequent values are decoded in constant time. This variant is particularly efficient if we are inside a run of differences equal to 1, as these are encoded using run-length encoding. Note that this only works for the initial range $[l, r]$, since the remaining accesses to Ψ are expected to be located at random and therefore they cannot be improved with this technique.

We also improve the strategy to binary search for $[l, r]$. We describe two alternative strategies, called *D-select+forward-check* and *D-select+backward-check*, which apply to patterns with 2 or 3 bound elements.

D-select+forward-check strategy

During a binary search in the *iCSA*, we compare the query pattern α with the string pointed by the current position in the suffix array, $T[A[i], n]$. The first steps of the binary search will be faster because the strings will differ in their first character, so the comparison will be decided with the first integer comparison without the need to compute Ψ, just $T[A[i]] = rank_1(D, i)$. At some step of the binary search, however, we will start to have $T[A[i]] = \alpha[1]$ and will have to compute $\Psi[i]$ in order to compare $\alpha[2]$ with $rank_1(D, \Psi[i])$; this access to Ψ can be relatively expensive if differentially compressed. Instead of performing all those isolated Ψ computations, in this strategy we perform all the checks for the complete range in order to filter the candidate positions.

Consider for instance the triple pattern (s, p, o), in which we would search for $\alpha = spo$. We first find the intervals that correspond to the subject, predicate, and object of the triple pattern: $R_s = [l_{s+gaps[0]}, r_{s+gaps[0]}]$, $R_p = [l_{p+gaps[1]}, r_{p+gaps[1]}]$, and $R_o = [l_{o+gaps[2]}, r_{o+gaps[2]}]$, using *select* operations on D: $l_c = select_1(D, c)$ and $r_c = selectnext(D, l_c)$ 1. Since Ψ is increasing within each of those intervals, we use these ranges to check, for each i in R_s, whether $\Psi[i]$ 2 R_p. Only a smaller range R_{sp} R_s will pass this filter, and the Ψ values in that range form in turn a range R_{ps} R_p. On this range R_{ps} we compute all the Ψ values to finally find the range R_{pso} R_{ps} of the values that map inside R_o by Ψ. Those are the final answer.

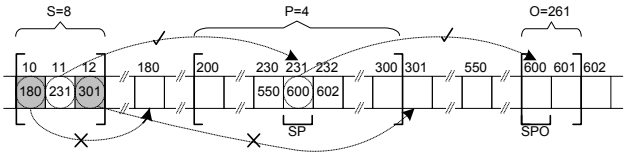

Fig. 4 D-select+forward-check strategy for pattern $(s; p; o) = (8; 4; 261)$.

Figure 4 shows an example of this operation. In this example, $R_s = [10, 12]$, $R_p = [200, 300]$, and $R_o = [600, 601]$. Checking the values of Ψ for the range R_s, we find that $\Psi[10]$ and $\Psi[12]$ do not map into range $[200, 300]$, but $\Psi[11]$ does. Therefore, we need to check if $\Psi[\Psi[11]]$ maps into the range $R_o = [600, 601]$, corresponding to object 261. Since it matches, we can report an occurrence of the triple $(8, 4, 261)$, i.e., confirm that the triple is in the collection.

In practice, this technique may be faster than a standard binary search if the initial interval (R_s in our example) is small enough. Note that, since our Ψ is cyclic, we can use any of the three intervals R_s, R_p, or R_o to begin our check. Typically, the number of objects is higher than that of subjects, so we expect that $jR_oj < jR_sj$ jR_pj. We may, however, choose on the fly the one that is actually shortest.

This strategy can also be applied to triple patterns with one unbound term. In this case, we perform the same operations but restricted to the bound terms. Assuming our bound variables are x and y, we compute R_x and R_y and perform the same range check to verify if, when applying Ψ to the positions in R_x, we end up in range R_y. Again, the cyclic nature of Ψ allows us to perform the range check independently of the position of the bound variables in the triple pattern. For example, for $(?s, p, o)$ triple patterns 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 $x = s$, $y = p$.

D-select+backward-check strategy

This strategy is based on the same ideas of the previous forward-check strategy. It relies on the fact that all positions i in R_s that pass the forward-check in the previous strategy necessarily form a subinterval of R_s. This means that, in order to discard candidate positions, we do not need to verify every i 2 R_s; instead, we can binary search for the subrange of positions that map to a valid range in R_p.

To take advantage of the previous property, we follow a similar idea to the well-known backward-search strategy [54]. Assume that we are searching for a triple pattern (s, p, o). We start our search now in interval $R_o = [l_o, r_o]$; since Ψ must be increasing within interval $R_p = [l_p, r_p]$, we binary search inside R_p in order to locate the subinterval $R_{po} = [l_{po}, r_{po}]$ R_p that contains all the positions i such that $\Psi[i]$ 2 R_o. If the subinterval is empty, no result exists for the query and we return immediately. Otherwise, we continue the backward-search process, binary searching in R_s in order to locate the subinterval $R_{spo} = [l_{spo}, r_{spo}]$ R_s that contains all the entries i 2 R_s such that $\Psi[i]$ 2 R_{po}. At

the end of this step, the range R_{spo} contains all the results for our query. Note that, when using an (s, p, o) pattern, either 0 or 1 results may arise, but we generalize this strategy to other triple patterns below.

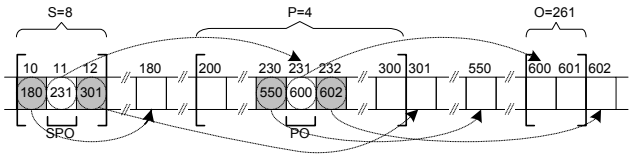

Fig. 5 D-select+backward-check strategy for pattern $(s; p; o) = (8; 4; 261)$.

Figure 5 displays an example of this strategy for a sample (s, p, o) query pattern. We start the backward search in range $R_o = [600, 601]$. Then we perform a binary search in the interval $\Psi[200, 300]$, in order to locate the subinterval that contains values that map into R_o; in our example, only the entry $\Psi[231]$ maps into $[600, 601]$, so we obtain a subinterval $R_{po} = [231, 231]$. Next, we continue the backward-search in R_s. We binary search inside the range $\Psi[10, 12]$ and locate the subinterval that maps to 231; in the example, only $\Psi[11] = 231$ maps. Consequently, the final interval is $R_{spo} = [11, 11]$, that contains the single occurrence for the given pattern.

This strategy can be easily adapted to work with all the query patterns that contain a single unbound variable. In $(s, p, ?o)$ queries, we locate the subinterval $R_{sp} \subseteq R_s$ that maps into R_p after applying Ψ. In $(s, ?p, o)$ queries, we locate the subinterval $R_{os} \subseteq R_o$ whose Ψ entries map into R_s. In $(?s, p, o)$ queries, we locate the subinterval $R_{po} \subseteq R_p$ whose entries map into R_o.

3.3 Supporting join operations

RDFCSA can be extended to support join operations by implementing different join techniques on top of the basic triple pattern query algorithms. We first describe the general technique, which can be used with any number of unbound elements in the triple patterns and for subject-subject, subject-object, and object-object join operations. We then briefly explain particular optimizations that are applied on top of the general technique.

Join operations in *RDFCSA* are essentially performed by following either a *merge-join* strategy or a *chaining* strategy.

The merge-join strategy considers each triple pattern separately. The join variable is treated as an unbound variable in both triple patterns. The two corresponding triple patterns are solved independently, therefore obtaining two lists of results. The final step scans the resulting lists to compute their intersection.[2] For instance, to compute $(s_1, p_1, ?x) \bowtie (s_2, p_2, ?x)$, we first compute the two triple-pattern queries $Q_1 = (s_1, p_1, ?o_1)$ and $Q_2 = (s_2, p_2, ?o_2)$. The results of Q_1 and Q_2 are then intersected by the O component to retrieve only

[2]Since the results returned by the *RDFCSA* for some triple patterns are not necessarily sorted by the desired element, a sorting step may be required prior to the intersection.

the values where $o_1 = o_2$. The same strategy can be applied to any combination of triple patterns, with simple adjustments depending on the number of unbound variables in each side.

The chaining strategy, instead, solves one of the triple patterns first, considering the join variable as unbound. Then, for each result obtained in this query, the second pattern is executed with the corresponding value of the join variable, which is now bound. The previous example, $(s_1, p_1, ?x) \bowtie (s_2, p_2, ?x)$, is executed following this strategy by first querying $(s_1, p_1, ?o_1)$, and then replacing each value o_1 obtained for $?o_1$ in the second pattern as (s_2, p_2, o_1). We speak of left-chaining if we start with the left triple pattern and apply each result as bound variables in the right one (as in the previous example), and of right-chaining if we start executing the right triple pattern and replace the results in the left one. The selection of the first pattern for chaining is important when the triple patterns have a different number of unbound variables.

In *RDFCSA* we have implemented a general mechanism to perform joins following the merge strategy as well as a left- or right-chaining strategy. Depending on the characteristics of the join, and particularly the location of the unbound variables, the strategy selected leads to significantly different triple-pattern queries, and therefore to important differences in query performance. The selection of the optimal strategy is a significant problem by itself. We test all possible strategies in our experimental evaluation, with one exception: strategies that would lead to the evaluation of an $(?s, ?p, ?o)$ pattern as a first step are not considered in any case, since decompressing the full dataset as an intermediate result would be very inefficient in terms of time and space.

3.3.1 Optimization of join operations

Some optimizations are added on top of the general join strategy, to take advantage of the characteristics of our technique and specific join patterns. These optimizations have a significant effect on the amount of computation performed by *RDFCSA* in most join operations.

The first enhancement to the basic algorithms is related to the dictionary encoding used. Recall that in the dictionary encoding used by HDT, all elements that are both subject and object are assigned an id lower than that of any element that only appears as a subject or as an object. This can be used to filter out results when performing subject-object joins. For example, to answer a query $(s_1, p_1, ?x) \bowtie (?x, p_2, o_2)$ using left-chaining, we would first obtain all the objects that match the triple pattern $(s_1, p_1, ?x)$; then, we have to check that each result matches the right triple pattern. However, with the dictionary encoding we use, we can immediately discard any result of the first query with an id higher than SO, since we know that it only appears as an object and therefore it will not match the overall join query. Note that this improvement is specific to this dictionary encoding, and is not specific to *RDFCSA*; the same optimization is also used, for instance, in K2Triples.

Another simple optimization that is applied to the merge strategy consists in taking into account the characteristics of the result list returned. In some

join patterns, we must sort both lists to compute their intersection; however, due to the evaluation mechanisms of *RDFCSA*, in some triple patterns the list of results is already sorted. For instance, the $(s, p, ?o)$ triple pattern returns a sorted list of objects as a result; therefore, to answer a query $(s_1, p_1, ?x) \bowtie (s_2, p_2, ?x)$, we can execute the two triple-pattern queries and then simply intersect the corresponding sorted lists. A similar idea is also applied to the chaining strategy: we can avoid some computation in the chaining phase by identifying repeated results. In order to do this, we sort the results of the first triple pattern and skip the computation of the second triple pattern on the repeated results of the first query. Therefore, we build the results of the final join only from the non-repeated results of the first triple pattern.

An additional improvement we include in all our join operations, when possible, is *variable filling*. As explained before, when running most triple-pattern queries, we first obtain the location of the set of triples and then use Ψ to retrieve the missing variables in the triple. This cost is necessary to return the complete result in a triple-pattern query. However, in join queries that follow the merge or chaining strategy, many of the matches found in the first pattern may not correspond to valid results of the overall join operation, since they do not have a match for the join variable in the second pattern. Our algorithms identify, depending on the type of join and the evaluation strategy, which variables in a triple pattern are necessary to solve the join and which ones are only necessary to make up the final result. The latter variables are filled in only after the complete join has been evaluated. We then use slightly modified versions of each triple-pattern query, customized according to which of the elements in the resulting triple have to be computed. The general algorithms solve the join using the incomplete triples (hence avoiding the rather costly Ψ computations on non-sampled positions, and rank operations), and then take care of refilling the missing variables once the join has been completed.

For instance, to perform the join $(?s_1, p_1, ?x) \bowtie (?x, p_2, o_2)$ with left-chaining, the first step is to compute the left triple pattern $(?s_1, p_1, ?o_1)$. This is usually done by first locating the range of p_1, and then using Ψ to locate the corresponding objects, and Ψ again to get the subjects. However, for the join operation we do not need the subjects, only the objects, so we do not compute the subjects yet: we first complete the join query, and then fill in the missing subjects for the resulting tuples.

4 Experimental evaluation

4.1 Experimental framework

We tested the compression and query performance of our proposal using the *DBPedia* dataset,[3] "the nucleus for a Web of Data" [6]. The original size of the dataset is around 34GB. It contains 232,542,405 triples in total, 18,425,128 different subjects, 39,672 different predicates, and 65,200,769 different objects.

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

After applying dictionary encoding to the triples, the structural part of the dataset can be stored in 2,790,508,860 bytes, using three 32-bit integers per triple.

We compare *RDFCSA* with K2Triples and permuted trie indexes, as good examples of other well-known state-of-the-art solutions that are similar to *RDFCSA*, in the sense that they are based on compact data structures and designed to work with triples composed of integer identifiers. We also compare our proposal with a number of alternative solutions following other approaches: HDT, Tentris, Virtuoso (open source edition, version 7.2.5.1), Blazegraph (version 2.1.4), MonetDB (version 1.7), and RDF-3X (version 0.3.7). Note that all of the latter can handle the RDF datasets in their original form as string triples. For HDT, we display in the plots the space required only for the Triples component, so it is directly comparable to *RDFCSA*, K2Triples, and permuted trie indexes. The same occurs for MonetDB, where we also store and query integer identifiers instead of strings. For Tentris, Virtuoso, Blazegraph, and RDF-3X we display the full size of the structure, after loading the original RDF dataset. The effect of the dictionary in space and query times will be discussed later.

Regarding query times, measurements are also taken differently in each family of solutions. For Virtuoso, Blazegraph, Tentris, MonetDB, and RDF-3X we measure query times using the utilities provided by each tool, which includes, in general, the cost of parsing the query. For *RDFCSA*, K2Triples, permuted trie indexes, and HDT, we measure the performance of queries on the integer ids, therefore ignoring any additional costs associated to the query tool and the SPARQL query parsing necessary in the other solutions. Query times are always displayed in μs/result to reduce the effect of the overhead required by the more complex tools.

For *RDFCSA*, we test the different algorithms and variants using different sampling intervals on Ψ, t_Ψ *2 f*$4, 8, 16, 32, 64, 512$*g*, so as to obtain a wide space/time tradeoff. Additional details on the variants and configurations will be given later.

For K2Triples we use the settings recommended by the authors. We use both the original K2Triples and the improved K2Triples+ that includes extra indexes to speed up queries with unbound predicate.

We test two configurations of the permuted trie index:[4] *trie-3t* and *trie-2tp*. The former has better performance and offers more stable query times because it is efficient over all triple patterns. Instead, *trie-2tp* uses only two of the three permutations, so as to reduce space while maintaining query times in most triple patterns. The main drawback of *trie-2tp* is that it performs much worse on $(?s, ?p, o)$ queries. There are other configurations of the permuted trie index, but we have chosen the best performing ones according to its authors.

For HDT, we use the original implementation by the authors.[5] To provide comparable query times, we performed minimal changes to the source code in

[4] https://github.com/jermp/rdf_indexes
[5] http://www.rdfhdt.org/

order to measure only the structural part of the query. To do this, we precompute the string-to-id translation for all queries, and then measure query times to return all results as identifiers, omitting the final id-to-string translation that is usually performed to return the final results. Therefore, our plots reflect the space and time required to solve the query on ids, omitting the space and time required for the HDT dictionary.

For MonetDB,[6] we store the integer ids corresponding to the triples to make its results directly comparable to the previous solutions. We executed queries using the mclient command-line tool, and use the query times it reported.

For Virtuoso,[7] we use the ingestion and query tools provided with the software. Particularly, we choose the interactive command-line query tool isql to execute queries, and use the query times it reported. Note that Virtuoso includes a server that provides an HTTP endpoint that can be used to run SPARQL queries. We have also tested query times in this interface, but the overhead caused by this endpoint was very significant (query times were 1.3–3 times larger than in the command-line tool); additionally, the HTTP endpoint limits the number of results returned, making it impractical for our purposes.

For Blazegraph,[8] we use a custom Java program that connects to Blazegraph in embedded mode. Query times are measured using `System.nanoTime`. The SPARQL endpoint provided by Blazegraph was also tested, but the overhead caused by it was also significant.

For Tentris, we use the query tool tentris_terminal,[9] provided by the authors. We display query times as measured by the tool. Since parsing times are disgregated by the tool, we display two different times for Tentris: `Tentris` represent total times, whereas `Tentris-noparse` exclude the parsing time from the total. Again, we have tested the HTTP endpoint provided, but we omit these results in our plots because they were up to 10 times worse than the command-line results in some queries.

In RDF-3X,[10] we use the command-line query tool provided to run the queries and measure query times.

We use an existing testbed for the DBPedia dataset.[11] This query set provides 500 queries for each of the 7 basic triple patterns, and 25 queries for each join pattern considered (additional details on the join variants and their classification will be provided in Section 4.4). For a fair comparison with tools that require access to disk, we execute a warm-up phase before running each query set. The warm-up includes performing the full set of triple pattern queries. After that, we execute each query set, measuring query times. Additionally, we set a number of repetitions of the full query set for triple pattern queries to guarantee accurate average time measurements.

[6] https://www.monetdb.org/
[7] https://virtuoso.openlinksw.com/
[8] https://blazegraph.com/
[9] https://github.com/dice-group/tentris
[10] https://code.google.com/archive/p/rdf3x
[11] Provided by the authors of K2Triples, available at http://dataweb.infor.uva.es/queries-k2triples.tgz

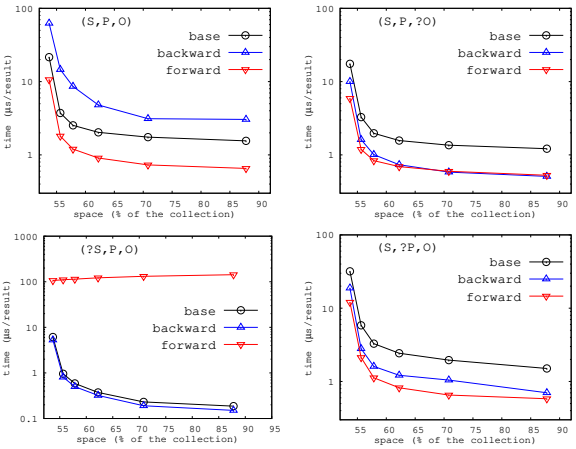

Fig. 6 Query times of the search variants on query patterns with zero or one unbound variable. Times in microseconds per result returned and in log scale.

We ran our experiments on an Intel Xeon E5-2470@2.3GHz (8 cores) CPU, with 64GB of RAM. The operating system was Debian 9.8 (kernel 4.9.0-8-amd64). The version of GCC was 6.3.0, and the version of Java (used to run Blazegraph) was 1.8. Our code, as well as the source code for RDF-3X and HDT, were compiled using GCC, with full optimizations. The remaining tools were installed using the packages/binaries provided by the authors. We have made our source code available at https://lbd.udc.es/research/rdf/.

4.2 Comparison of the query algorithms of *RDFCSA*

First we analyze the relative performance of the query algorithms developed for our structure, discussed in Section 3.2. We measure space and query times for the different triple patterns using the basic binary search algorithm (*base* in the plots), the D-select-forward-check strategy (*forward*), and D-select-backward-check (*backward*).

Figure 6 displays the space and query times for the different search algorithms.[12] We only show results for query patterns with zero or one unbound variable, because triple patterns with a single fixed variable lead to patterns α of length 1, where backward- or forward-check strategies cannot be applied. For the backward- and forward- strategies we use our *selectnext* optimization.[13] As shown in the figure, the baseline binary search is in general slower than the other alternatives. A notable exception occurs in $(?s, p, o)$ queries, where the forward-check strategy is very inefficient. This difference is due to the large number of occurrences that may have to be sequentially checked in R_p. Therefore, even though *D-select+forward-check* is faster in most cases, *D-select+backward-check* is in general more consistent. Note, nevertheless, that

[12] The space is given as a percentage of the size of the raw data, which for this purpose is taken as a binary representation of the triple patterns with each triple stored using three 32-bit integers.

[13] Further details comparing *select* implementations will be given in Figure 7.

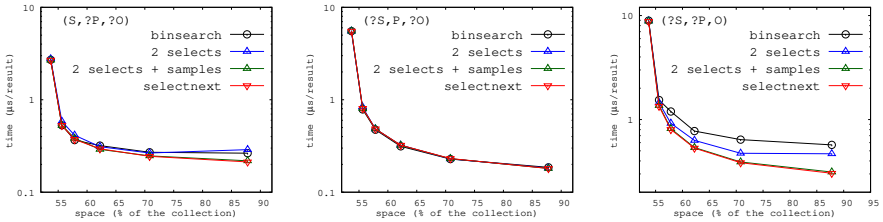

Fig. 7 Comparison between basic binary search in the *iCSA* and dual select for patterns with one fixed term. Times in microseconds per result and in log scale.

we can easily select the best algorithm for each triple pattern, and we can even perform on-the-fly selection of the best query algorithm using a simple heuristic depending on the length of the ranges involved. For simplicity, in the following experiments we only display the query time of the most efficient search technique in each query pattern (i.e., *D-select+forward-check* in most cases, *D-select+backward-check* in $(?s, p, o)$ queries). Note also that the results presented in this section are those of the basic implementation of *RDFCSA*. Additional plots are omitted for simplicity, but we have obtained similar results for other implementation variants, with *D-select+backward-check* being the most consistent search strategy overall.

Next, we analyze the impact of our improvements on $select_1$ queries on triple patterns with two unbound variables. In these queries, we must search for a pattern α of length 1, so we can replace the standard binary search of the *iCSA* by two *select* operations in Ψ to locate the appropriate interval $[l, r]$. Further, the second *select* can be replaced with the *selectnext* algorithm, which is faster (see Section 3.2.2).

Figure 7 displays the performance of the binary search on Ψ (*binsearch*), of replacing it with two $select_1$ operations on D implemented with binary searches (*2 selects*), of improving those $select_1$ operations with sampling (*2 selects + samples*), and of replacing the second such $select_1$ with a *selectnext* operation (*selectnext*). The results show that each improvement makes a significant difference with the previous version, except for the use of *selectnext*, whose improvement is marginal but still always positive. Recall that we store the $select_1$ answers directly on D_p, thus in the triple pattern $(?s, p, ?o)$ there is no difference between *binsearch* and the various $select_1$ variants. Considering these results, in the remaining experiments we will always use the *selectnext* algorithm when applicable.

4.3 Comparison with other RDF representations

In this section we compare *RDFCSA* with state-of-the-art alternatives. We start by measuring their space requirements and query performance on simple triple patterns. We show compression as a percentage of the original size of the collection (considering an integer-base representation). We test three implementation variants of *RDFCSA*. In all of them, we use the algorithms that obtained the best results in previous tests: *selectnext* to obtain ranges using

D, *D-select+forward-check* for most patterns that require search on Ψ, and *D-select+backward-check* for $(?s, p, o)$ patterns. The three variants of RDFCSA tested are the following:

- *RDFCSA* is the basic implementation[14], with D and Ψ partitioned into three arrays. Those for D are bitmaps in plain form with $rank_1$ [28] and our faster $select_1$ structures, yet D_p stores the $select_1$ answers in plain form. The Ψ arrays are compressed with Huffman and run-length encoding (RLE) [21].
- *RDFCSA-rrr* is like the basic variant but the bitmaps of D are compressed using the RRR technique [53] with sampling parameter 128.
- *RDFCSA-Hybrid* is the *hybrid* variant, with Ψ_s and Ψ_o stored as plain arrays where entries use $d\log_2 ne$ bits, and Ψ_p compressed as usual with Huffman and RLE.

Figure 8 shows the space/time tradeoffs obtained by all the solutions in the core triple-pattern queries. We display a plot per triple pattern, including the values for each alternative.

Let us first focus in the comparison among our *RDFCSA* variants. The *RDFCSA-rrr* variant, which aims at reducing the space of *RDFCSA*, is moderately successful in that sense, with little impact in the time when the structures use little space (i.e., nearly 50% of space thanks to a sparse sampling of Ψ). Thus, it is an interesting alternative to reduce space. However, when we aim at improving the query performance by using a denser sampling of Ψ, the *RDFCSA-rrr* becomes much slower than the basic *RDFCSA*. The *RDFCSA-Hybrid* variant, instead, uses at least 65% of space, but it is significantly faster than the basic *RDFCSA*. This variant improves its times with a denser sampling of Ψ only in query patterns where the subarray Ψ_p is involved.

We next focus on the comparison with other solutions. The results show that *RDFCSA* requires more space than K2Triples, and even than the faster K2Triples+. The trie-based solutions achieve significantly different compression rates: *trie-2tp* is comparable in space to *RDFCSA*, whereas *trie-3t* is up to 60% larger. MonetDB and HDT are also close to the compression ratio of *RDFCSA*, whereas the remaining alternatives require significantly more space: Virtuoso and RDF-3X require 7–8 times the space of *RDFCSA*, Blazegraph is 10 times larger and Tentris is 20 times larger (note that a triple break is added to the x axis, to display all results together, distorting the huge differences in space between these techniques).

In addition to being much larger, Virtuoso, Blazegraph, and RDF-3X are much slower in general than the alternatives based on compact data structures. Note, however, that query parsing time is included in the measurements for these tools. In the case of Tentris, we display query times both including parsing time and excluding it, as this information is segregated by the query tool.

[14]The preliminary version of *RDFCSA* presented in [11] is essentially the new variant *RDFCSA* discussed here, yet it had non-partitioned *D* and Ψ arrays and did not include the *selectnext* optimization. In practice, the new *RDFCSA* improves the space of the former counterpart by around 1-2% and yields a similar performance, except for operations (S,P,O), (S,P,?O), and (S,?P,O) where our *selectnext* optimization leads to 1.1-1.9x faster query times.

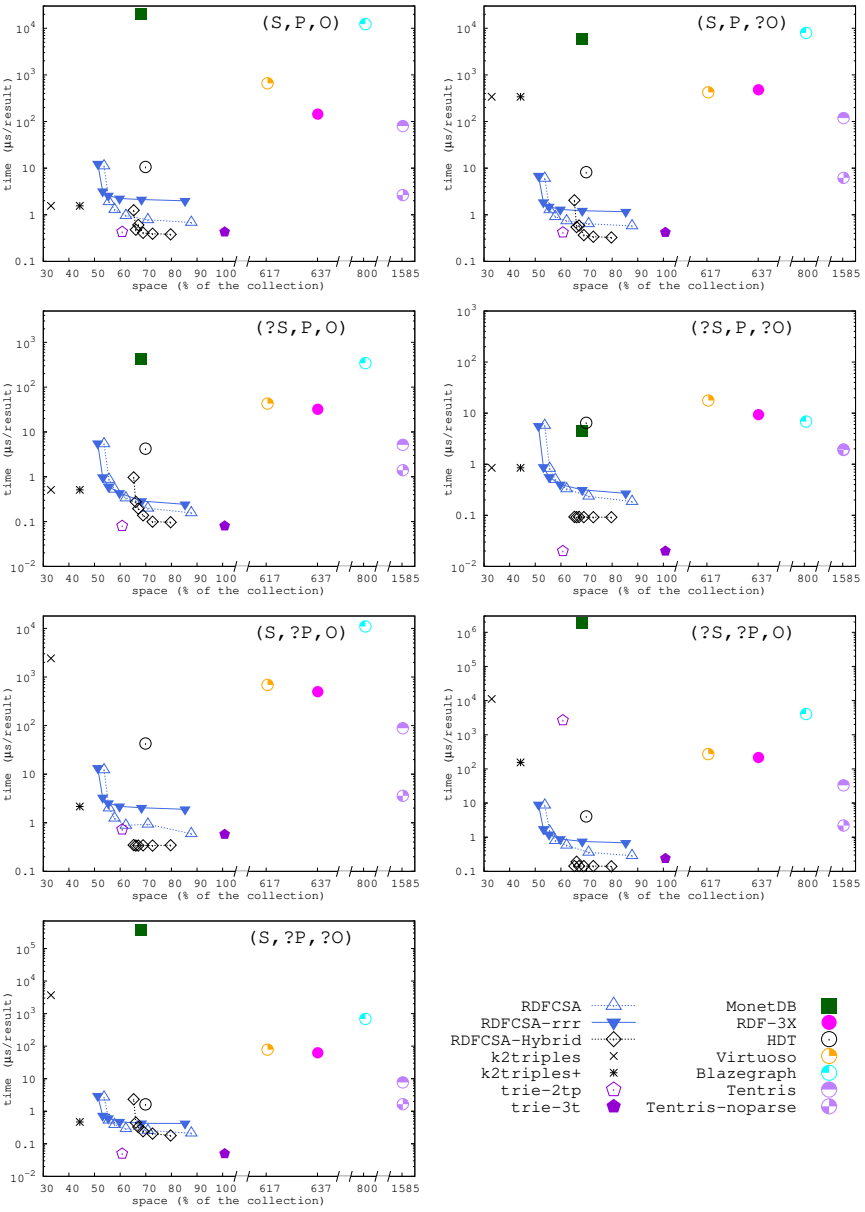

Fig. 8 Space/time tradeoff on the triple-pattern queries. Note the log scale in the y axis. The x axis uses linear scale but includes three breaks to include all results. Query times in microseconds per result reported. MonetDB did not completed in reasonable time for $(s; ?p; o)$ triple patterns.

Results show that parsing time causes a significant overhead in these queries, and ignoring this parsing time makes Tentris competitive in query times with our solutions, although using much more memory. Among the more compact

solutions, the hybrid *RDFCSA* yields the fastest query times in most patterns, improving on the performance of K2Triples and achieving query times competitive with permuted trie indexes: *RDFCSA* is competitive with *trie-3t*, requiring less space, and is more consistent than *trie-2tp*. HDT is easily dominated by *RDFCSA* variants in all query patterns.

Recall that we display the space and query times required to store and query triples of integers for the approaches based on compact data structures, but RDF-3X, Virtuoso, Blazegraph, and Tentris process the original RDF data. Space results are therefore not directly comparable, but these techniques are still a relevant baseline as SPARQL query tools. Note that *RDFCSA*, K2Triples, and permuted trie indexes could be complemented with a compact string dictionary that follows the encoding proposed for HDT. Solutions like HashDAC-RP [43] can answer string-to-id and id-to-string translations in a few microseconds per operation (typically requiring 1–4 microseconds per operation in URI and literal dictionaries such as those required in DBpedia [13, 43]). This dictionary would increase the size of the structure by an extra 60% of the collections in our plots, keeping them in roughly 90–150% of the original collection (still 4 times smaller than Virtuoso, the most compact of the alternatives). This means that, even adding the space required for such a dictionary, *RDFCSA* would still easily overcome Virtuoso, RDF-3X, Blazegraph, and Tentris in space. Additionally, since each triple-pattern query requires at most 3 string-to-id translations per query, and at most 3 id-to-string operations per returned result (at most 2 translations in practice, ignoring the $(?s, ?p, ?o)$ triple pattern), query times would be increased by less than 10μs per result in most cases when adding this dictionary. Note, however, that query times for Tentris ignoring parsing time (`Tentris-noparse`) are also below this limit, making it competitive in practice with *RDFCSA*. Virtuoso and Blazegraph are probably affected in similar amounts by parsing overheads in these queries, making them look less competitive than they could be in practice. In Section 4.4 we will show results for the more complex join operations, where the effect of the dictionary and query parsing overheads is less significant in general, and query times comparisons will be fairer.

We now discuss specific results for each triple pattern, though overall trends can be easily detected: K2Triples and K2Triples+ are the most space-efficient solutions, but their performance is difficult to assess, since it varies significantly among triple patterns. In turn, *RDFCSA* obtains consistently low query times, never exceeding 10 microseconds per result in any triple pattern for reasonable sampling intervals. *Trie-2tp* obtains compression comparable with that of *RDFCSA* and better query times in most triple patterns, yet as explained before it has a major drawback: the $(?s, ?p, o)$ pattern is up to 10,000 times slower than the others, and roughly 1000 times slower than *RDFCSA*, effectively limiting the application of this solution. The strongest counterpart, *trie-3t*, on the other hand, achieves the best query times in some cases, yet at the cost of much worse compression (*RDFCSA-Hybrid* outperforms it in the others, using less space). HDT is consistent in query times, but slower

and larger in general than *RDFCSA*. MonetDB is several orders of magnitude slower than *RDFCSA*, using similar space, whereas Virtuoso, Blazegraph, RDF-3X, and Tentris are much larger than our technique. Query times for Virtuoso, Blazegraph, and RDF-3X are still much higher than those of *RDFCSA* in general. Results for Tentris, however, show that for most of the triple patterns the cost of query parsing is much larger than the query execution itself, that only requires a few microseconds per result. This is comparable to *RDFCSA*, that would still have to be augmented with a dictionary to transform integer IDs in the result to the original strings. Nevertheless, Tentris requires over 20 times the RAM of *RDFCSA* (even augmenting *RDFCSA* with the string dictionary, Tentris would still be 10 times larger), so we do not consider it to be a fair competitor for *RDFCSA* and the other compact solutions.

Therefore, in what follows we focus on the comparison between *RDFCSA*, K2Triples, and *trie* variants. We will resume the comparison with the remaining triple stores when testing join queries, in which the relative overhead of query parsing should be much smaller, and solutions like Virtuoso and Blazegraph become more competitive.

The simplest triple pattern, (s, p, o), is the best case for K2Triples, since it performs a single-cell retrieval query at (s, o) in the k^2-tree associated with predicate p. In terms of time per result, this query is the worst for *RDFCSA*, since it searches for a pattern of length 3 to return at most one occurrence. Still, *RDFCSA* outperforms K2Triples with a reasonable sampling for Ψ (i.e., using over 55% space). The variant *RDFCSA-Hybrid* is the fastest, together with the *trie* variants. The situation is very similar for the triple pattern $(?s, p, o)$, where K2Triples has to scan a short column for fixed coordinate o in the grid.

K2Triples worsens by orders of magnitude in triple patterns $(s, p, ?o)$, because it has to scan all the objects in a long row (fixed s coordinate) of the k^2-tree associated with predicate p. Instead, *RDFCSA* and *trie* variants are almost unchanged. In fact, *RDFCSA-Hybrid* becomes slightly faster than the *trie* variants when using 70% space.

In the triple pattern $(?s, p, ?o)$, K2Triples simply retrieves all the points in the k^2-tree of predicate p, so its time per result is good (but still outperformed by *RDFCSA*). This time, the *trie* variants sharply outperform our fastest variant, *RDFCSA-Hybrid*.

The lower half of Figure 8 displays the three triple patterns where the predicate is unbound. In these patterns, K2Triples is very inefficient, so we compare with K2Triples+, which uses significantly more space (yet still less than *RDFCSA*). As before, even the basic *RDFCSA* outperforms K2Triples+ once using over 55% of space, by orders of magnitude on $(?s, ?p, o)$. Our fastest variant, *RDFCSA-Hybrid*, also outperforms the *trie* variants, except on $(s, ?p, ?o)$, where the latter are clearly faster. Note that the main drawback of *trie-2tp* shows on $(?s, ?p, o)$, where it is several orders of magnitude slower.

Overall, the results show that *RDFCSA* is an intermediate spot between K2Triples, which achieves by far the best compression among the tested solutions (but is outperformed in time by *RDFCSA*), and *trie-3t*, which disputes

the best query times with our variant *RDFCSA-Hybrid* (but uses more space). *RDFCSA* stands out as a very relevant space/time tradeoff, while offering stable and predictable times across all triple-pattern queries. This consistency is particularly significant taking into account that triple patterns are the basis for more complex SPARQL queries, which perform joins involving a number of triple patterns. An inefficiency in one triple pattern may sharply degrade the performance of the whole complex query. This is a problem in variants like *trie-2tp* and K2Triples+, which are several orders of magnitudes slower on some triple patterns, and makes them less appealing for a general-purpose SPARQL query engine.

4.4 Join queries

After analyzing *RDFCSA* on basic triple patterns, we study the performance of the different solutions in join queries involving two triple patterns. In this section we only display results for some of the relevant state-of-the-art alternatives used previously. Particularly, we keep K2Triples and K2Triples+, MonetDB, RDF-3X, Virtuoso, and Blazegraph. We exclude from this comparison HDT and permuted trie indexes, that have no specific mechanisms for joins, and implementing merging or chaining evaluation on top of their triple pattern queries would yield the same relative performance with respect to *RDFCSA* we observed in Figure 8. We also omit results for Tentris, since parsing errors were returned for most of the join queries in our query sets. Further, for simplicity we only display results for the basic implementation (*RDFCSA*) and the *Hybrid* version (*RDFCSA-Hybrid*). Finally, even though *RDFCSA* can still obtain space/time tradeoffs for join queries, for the sake of clarity we focus the analysis in this section on query times, and display results only for one sampling period of Ψ ($t_\Psi = 32$, the third point left-to-right in Figure 8).

We analyze the results for all the different binary join queries that can arise in practice, involving two triples, using an existing testbed [3]. Figure 9 displays the different query types included in the testbed and their characteristics. This testbed categorizes the joins by the number of unbound predicates, and the number of unbound subject/objects. For instance, join A has no unbound predicates, and no unbound subject/object. Therefore, a pattern for this join is $(s, p_1, ?x) \bowtie (?x, p_2, o)$. This is the subject-object variant of this join, since the join variable $?x$ is subject in one triple and object in the other. Two other variants of the same join can be created: $(?x, p_1, o_1) \bowtie (?x, p_2, o_2)$ (subject-subject join), and $(s_1, p_1, ?x) \bowtie (s_2, p_2, ?x)$ (object-object join). Figure 9 details the specific bindings for subject-object joins, but the remaining configurations can be easily inferred.

In the following sections we will display results categorized according to the number of unbound predicates in the join patterns. This has little effect on performance for *RDFCSA*, but severely affects tools based on vertical partitioning like MonetDB and K2Triples (although the K2Triples+ variant of K2Triples mitigates this problem with its extra indexes). In each category,

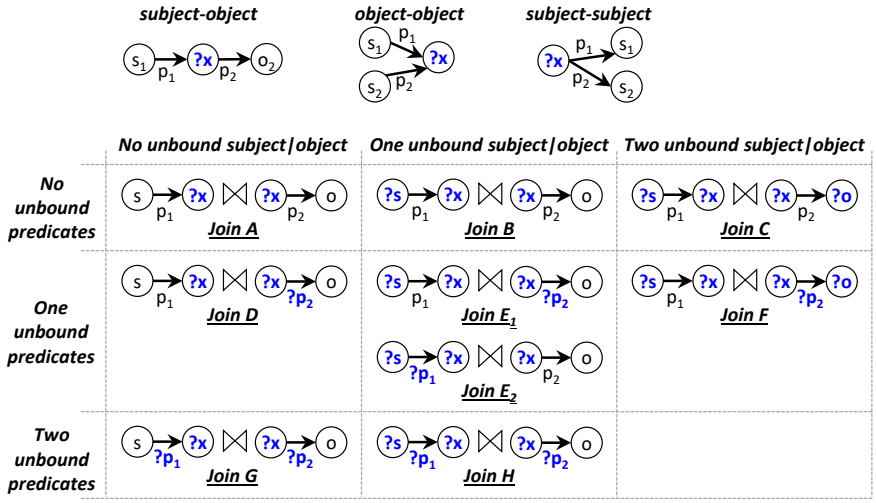

Fig. 9 Join variants included in the testbed. Joins are classified by number of unbound predicates, and number of unbound variables.

queries are listed in order of increasing "complexity", in the sense that additional unbound variables generally lead to a larger number of intermediate results, and therefore additional computation is required. For instance, joins A, B, and C have no unbound predicates, and have 0, 1, and 2 unbound subject/objects respectively, so join C should be more complex in general than join A.

The different configurations yield 9 join patterns (A, B, C, D, E.1, E.2, F, G, H), each with 3 variants: subject-subject (SS), subject-object (SO), object-object (OO). Following the original testbed, for each join type and variant we use two different query sets (-big and -small), which differ in the average number of results returned by the queries. This yields a total of 54 query sets.

Finally, for each join type, we display query times for the different join strategies applied in each case: merge-join (-merge), and left- (-left) and right-chaining (-right), as well as interactive evaluation in K2Triples (-int) [3]. Note that in some joins, specific strategies are inherently less efficient; we display all of them for *RDFCSA* in our results for completeness, excluding only the alternatives that would cause a full database query $(?s, ?p, ?o)$. Because of the inherent inefficiency of some techniques depending on the type of join, we will focus our discussion mainly on the most efficient strategies for each join type. Moreover, for some query patterns and configurations we were not able to obtain results in reasonable time with some tools: multiple query sets could not run in MonetDB, including all variants of join G and H, due to the two unbound predicates; several query sets are also omitted for RDF-3X, Virtuoso, and Blazegraph; a few query sets also failed with K2Triples or K2Triples+. When no time could be obtained, the corresponding bar will appear empty in the plots that display the results.

As discussed before, space and time comparisons between the more compact solutions (*RDFCSA* and K2Triples) and the remaining alternatives may be affected by SPARQL query parsing and other overheads that are not considered in the former. Therefore, the direct comparison of the results could be unfair to SPARQL-compliant stores, especially in the simpler joins that return a smaller number of results. In more complex joins, in which the number of unbound variables is large, and especially in -big query sets, the overhead of query parsing should be smaller and time comparisons should more accurately reflect the actual query performance. Taking all of this into consideration, we will focus most of our analysis on the comparison between *RDFCSA* and K2Triples (or K2Triples+), highlighting only particular cases where the performance of the other systems should be noted.

4.4.1 Joins with no unbound predicates

For these joins, we display results for K2Triples, since the additional indexes used by K2Triples+ do not yield any improvement in performance for fixed-predicate queries. The top plot of Figure 10 displays the results for join A (e.g., $(s, p_1, ?x) \bowtie (?x, p_2, o)$, with no unbound variables). In this join, *RDFCSA* with left chaining obtains the best results in all cases. This technique, for subject-object joins, essentially executes each join as an $(s, p_1, ?x)$ query chained with (x_i, p_2, o) queries for each x_i that results from the first query. The results are similar for object-object joins, but for subject-subject joins, K2Triples obtains better query times. This difference, depending on the position of the join variable, is consistent with our previous results on triple patterns: when executing an object-object or subject-object join with left chaining, the first query executed involves an $(s, p, ?o)$ pattern, where *RDFCSA* was two orders of magnitude faster than K2Triples. However, on subject-subject joins, the first query is an $(?s, p, o)$ pattern, where query times were similar. MonetDB, RDF-3X, Virtuoso, and Blazegraph are typically much slower than the best variant of *RDFCSA* by at least one order of magnitude. Note, however, that query parsing and other fixed overheads in these tools may be especially significant in these joins, that return a very small set of results.

The middle plot of Figure 10 displays results for join B (e.g., $(?s, p_1, ?x) \bowtie (?x, p_2, o)$, with one unbound variable). Several times, K2Triples obtains the best query times with its interactive evaluation strategy, but *RDFCSA-Hybrid* is the best in the other cases. The nature of this join, where one pattern has an extra unbound variable, leads to uncertainty in the complexity of the best operation order. Because of this, the interactive evaluation in K2Triples is a good approach, even though differences are usually small. MonetDB, RDF-3X, Virtuoso, and Blazegraph are competitive in some cases, especially in the -big executions in which the query parsing overhead is reduced and their ability to extract larger results sets is highlighted.

The bottom plot of Figure 10 displays results for join C (e.g., $(?s, p_1, ?x) \bowtie (?x, p_2, ?o)$, with two unbound variables). In this type of join, *RDFCSA* again obtains the best query times, usually with left-chaining evaluation. This is

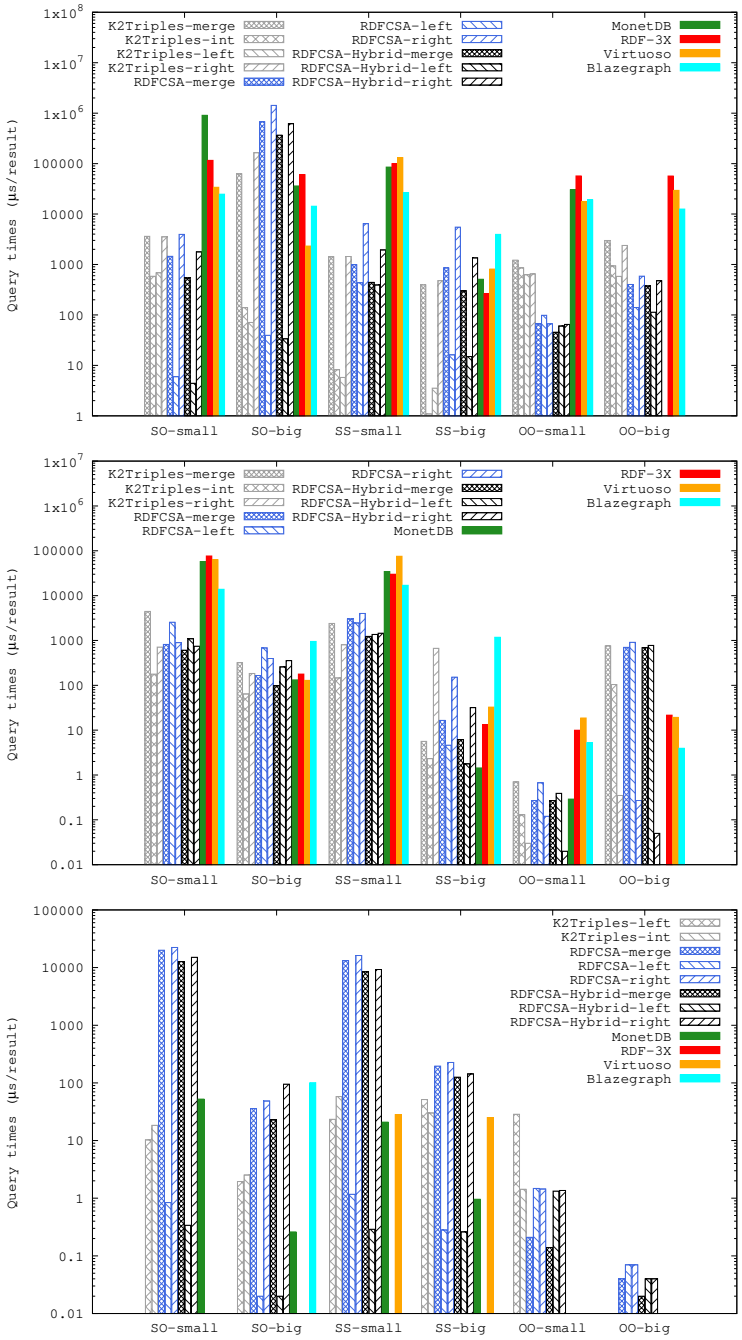

Fig. 10 Results for join A (top), B (middle), and C (bottom).

clearly the most efficient technique for this join, with results similar to those of join A. When both triple patterns have a similar structure (i.e., the same number of fixed and bound variables), *RDFCSA* tends to be more efficient with left-chaining, due to the performance of the triple-pattern queries that are generated: in subject-object joins, with left-chaining, we run an $(?s, p_1, ?o_1)$ query followed by many $(s_i, p_2, ?o_2)$ queries, which are very efficient in *RDFCSA*. However, in object-object joins the merge strategy is better. Regarding MonetDB, Blazegraph, and Virtuoso, we obtain similar results as for Join B (i.e. they are at least one order of magnitude slower than the best choice), yet we can see that in most cases we could not get results for those techniques.

4.4.2 Joins with one unbound predicate

Figures 11 and 12 display the query times for joins D, E, and F. In these experiments we compare *RDFCSA* with K2Triples+ instead of K2Triples, since the latter is typically orders of magnitude slower.

Considering the results across all the joins, *RDFCSA* achieves better query times. Yet, results are significantly different depending on the join type and query set. MonetDB is far from competitive as long as an unbound predicate appears, as expected, and it is up to 5 orders of magnitude slower than the other techniques. RDF-3X, Virtuoso, and Blazegraph are also 1–2 orders of magnitude slower than the fastest *RDFCSA* variant in most cases. Hence, we will focus on the comparison between *RDFCSA* and K2Triples+.

In join D, *RDFCSA* obtains the best overall results for object-object joins, but K2Triples+ is also competitive. K2Triples+ is faster in subject-subject joins and in some cases for subject-object joins. Left-chaining is the best strategy in most cases, both in K2Triples+ and *RDFCSA*, since it evaluates the triple pattern with bound predicate first, therefore saving a significant effort on the right triple pattern.

Regarding join F, *RDFCSA* is significantly faster in all cases, again with left-chaining, as this reduces the cost of processing the pattern with unbound predicate. Note that, for this join, most alternatives failed to yield results for the object-object joins in our setup. Finally, note that when comparing joins D and F, we find the same trend existing between joins A and C: K2Triples and K2Triples+ are more competitive with few unbound variables. In more complex queries, instead, *RDFCSA* is much more efficient.

Figure 12 shows significant differences between joins E1 and E2, because the different location of the unbound predicate leads to very different triple patterns in each side of the join. The join E1 (e.g., $(?s, p_1, ?x) \bowtie (?x, ?p_2, o)$) requires much more computation with any of the basic strategies, since both triple patterns contain an unbound variable. The best evaluation strategy is unclear: the merge and right-chaining techniques are competitive in *RDFCSA*, but K2Triples+ is slightly faster in most cases with its interactive evaluation. However, in join E2, the left pattern is much simpler than the right one, leading to a clearer evaluation path: left-chaining is the best strategy, and *RDFCSA* is an order of magnitude faster than K2Triples+ in most joins.

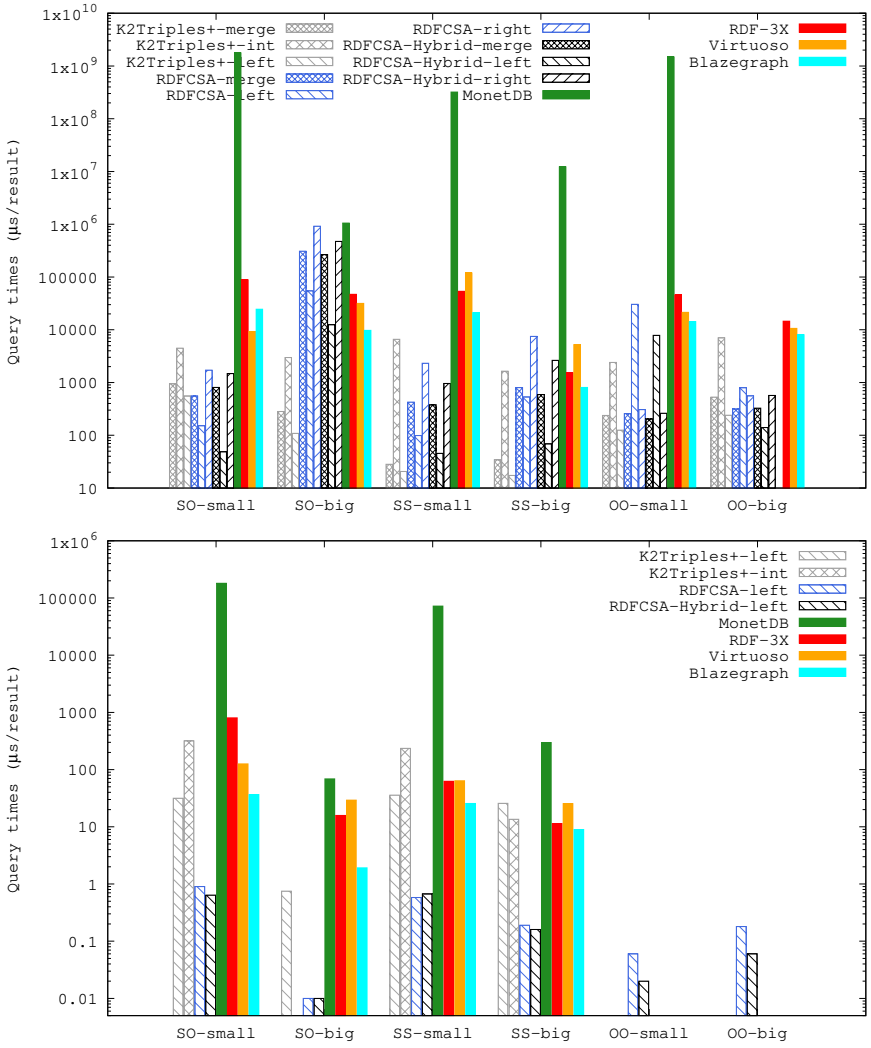

Fig. 11 Results for join D (top) and F (bottom).

4.4.3 Two unbound predicates

Figure 13 displays the query times for joins G (e.g., $(s, ?p_1, ?x) \bowtie (?x, ?p_2, o)$) and H (e.g., $(?s, ?p_1, ?x) \bowtie (?x, ?p_2, o)$). We omit MonetDB in these joins because the combination of two unbound predicates makes those queries extremely inefficient in its vertical partitioning model.

Like in previous cases, the results vary significantly depending on the join and query set. For join G, *RDFCSA* is the fastest technique in almost all cases, using merging or left-chaining depending on the case. For join H, *RDFCSA* with right-chaining is also orders of magnitude faster than K2Triples+ in general. These results are again consistent with the trend in previous sections that

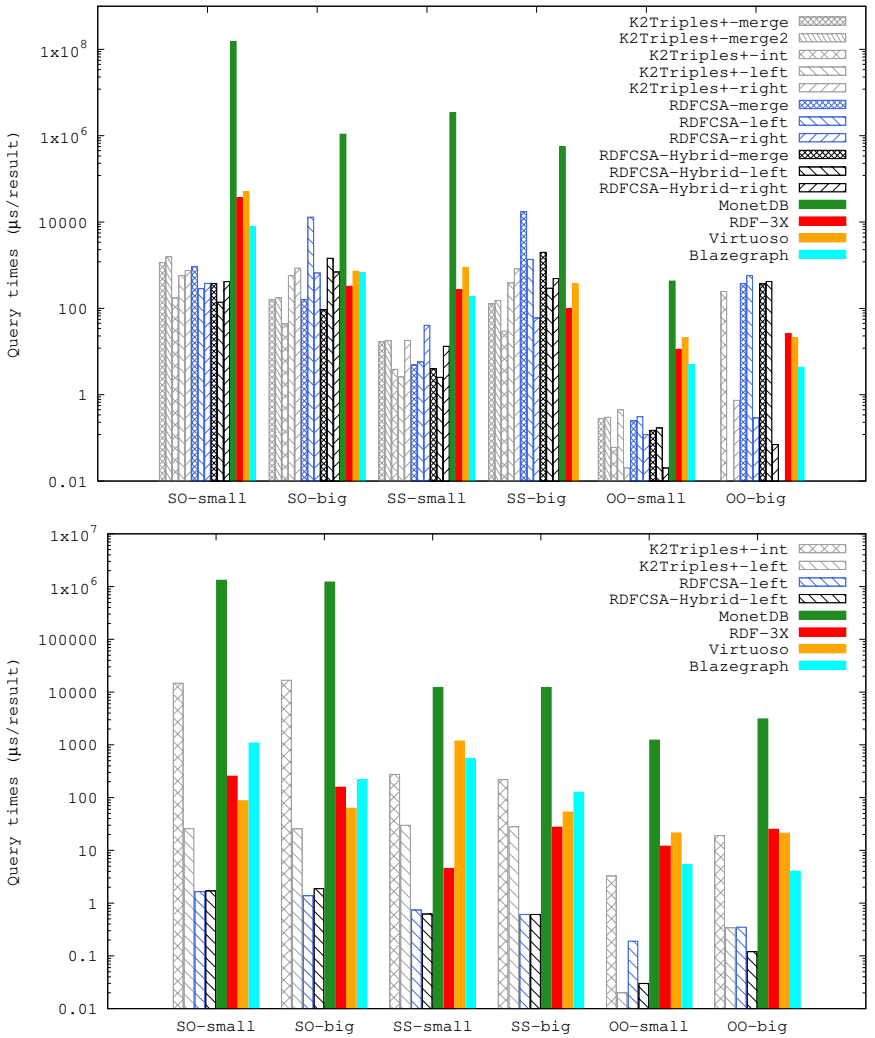

Fig. 12 Results for joins E1 (top) and E2 (bottom).

suggests that *RDFCSA* is especially competitive in the more complex join patterns. The subject-subject joins with many results are the only observed case where RDF-3X, Virtuoso or K2Triples are faster than *RDFCSA*. In most other query sets, however, the fastest *RDFCSA* variant is 1–2 orders of magnitude faster than the other alternatives.

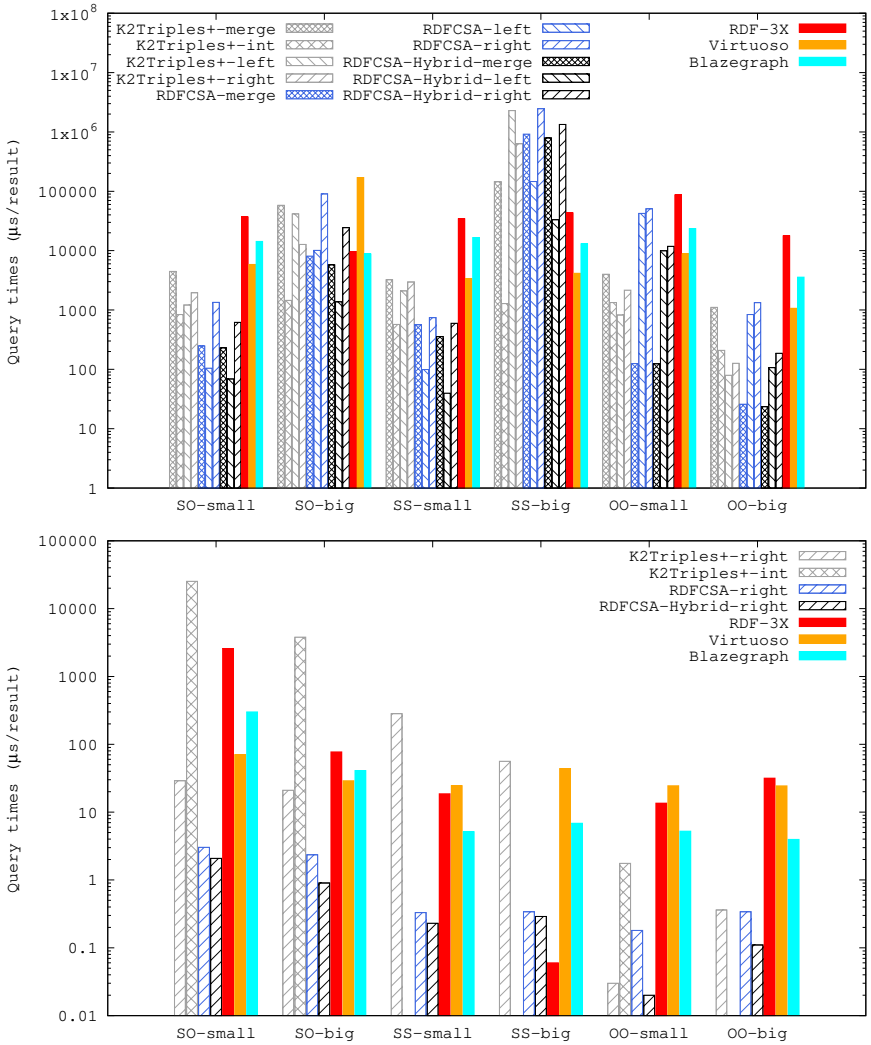

Fig. 13 Results for joins G (top) and H (bottom).

5 Conclusions

We have introduced *RDFCSA*, a compact data structure for the efficient storage and querying of RDF datasets. It is based on a compressed text index, the *CSA* [54], which is adjusted so that the triples that compose the RDF dataset are regarded as circular strings of length 3. We demonstrate that all the SPARQL triple patterns boil down to text searches in this particular collection of cyclic strings. The basic capabilities of *RDFCSA* are then based on the *CSA* search algorithms, which we have adapted and optimized for our scenario. We also design algorithms to solve queries involving joins.

RDFCSA is able to compress a set of RDF triples to around 60% of their raw size. Within this space, it offers fast and very consistent query times for all the basic triple-pattern queries, which are the basis for SPARQL support. In our experiments, *RDFCSA* answers any triple-pattern query within a few microseconds per result. It is also able to efficiently answer queries involving binary joins, being faster in most cases than the alternatives. Our experimental evaluation shows that state-of-the art solutions like RDF-3X, Virtuoso or Blazegraph are much larger, and in many cases slower than *RDFCSA*, even considering effects such as query parsing and dictionary encoding. We also clearly outperform HDT [23] in both space and time. Modern in-memory alternatives such as Tentris can achieve competitive query times with our solution, but their memory requirements are an order of magnitude higher than ours.

While K2Triples [3] obtains better compression than *RDFCSA*, its query times are much less consistent, being several orders of magnitude slower in some triple-pattern queries. The recent permuted trie indexes [51], on the other hand, are able to outperform *RDFCSA* in time, but in order to achieve consistent performance for all triple patterns they need to use around 50% more space. Our implementation variants also provide a wide space/time tradeoff, that can be easily tuned by adjusting the sampling interval on Ψ.

Overall, *RDFCSA* provides a very appealing space/time tradeoff for the storage of RDF data, combining low space with fast and consistent query times. Such predictability is very important when building up more complex SPARQL queries on top of simple triple patterns and joins.

Our current implementation is designed to handle integer-based triples, so it requires an external dictionary to handle the mapping between strings and ids. As future work, we plan to integrate *RDFCSA* with some compressed dictionary [13, 42, 43] in order to provide efficient mappings. Another choice is to integrate it in the HDT library (http://rdfhdt.org), which already provides the needed string dictionaries. Another future challenge is to make *RDFCSA* dynamic, that is, allow adding and removing triples from the database. This is already supported by indexes like RDF-3X and solutions like Virtuoso and Blazegraph; a dynamic implementation of K2Triples also exists [12]. We believe that it is possible to build on dynamic variants of the *CSA* [15, 38, 45] to obtain an efficient dynamic *RDFCSA*. Finally, compressed indexes inspired in the *RDFCSA* have been used to implement multi-join algorithms in worst-case-optimal time [4, 33], which for complex queries using the same variables several times are more efficient than query plans based on binary joins.

Acknowledgments. Funding for the Spanish group: projects funded by MCIN/ AEI/10.13039/501100011033: PDC2021-121239-C31 (FLATCITY-POC)-"NextGenerationEU"/PRTR; PDC2021-120917-C21 (SIGTRANS)-"NextGenerationEU"/PRTR; PID2020-114635RB-I00 (EXTRACompact); PID2019-105221RB-C41 (MAGIST); PID2021-122554OB-C33 (OASSIS-UDC); and TED2021-129245B-C21 (PLAGEMIS-UDC); grant ED431C 2021/53 (GRC) funded by GAIN/Xunta de Galicia; and grant ED431G 2019/01 (CSI) funded by Xunta de Galicia, FEDER Galicia 2014-2020 80%,

SXU 20%; Gonzalo Navarro is partially funded by Fondecyt 1-200038, and by ANID – Millennium Science Initiative Program – Code ICN17 002.

Data availability. The dataset DBPedia used in our experiments is available at http://downloads.dbpedia.org/3.5.1/. A processed version of the dataset, containing only the integer ids used by our representation, has been made available at https://lbd.udc.es/research/rdf/. The source code of *RDFCSA* and execution scripts are also available at the same url. The testbed used is available at http://dataweb.infor.uva.es/queries-k2triples.tgz.

References

[1] Abadi DJ, Marcus A, Madden SR, et al (2007) Scalable semantic Web data management using vertical partitioning. In: Proc. 33rd International Conference on Very Large Data Bases (VLDB), pp 411–422, https://doi.org/10.5555/1325851.1325900

[2] Ali W, Saleem M, Yao B, et al (2022) A survey of RDF stores & SPARQL engines for querying knowledge graphs. The VLDB Journal 31(3):1–26. https://doi.org/10.1007/s00778-021-00711-3

[3] Álvarez-García S, Brisaboa N, Fernández JD, et al (2015) Compressed vertical partitioning for efficient RDF management. Knowledge and Information Systems 44(2):439–474. https://doi.org/10.1007/s10115-014-0770-y

[4] Arroyuelo D, Hogan A, Navarro G, et al (2021) Worst-case optimal graph joins in almost no space. In: Proc. 2021 International Conference on Management of Data (SIGMOD/PODS), pp 102–114, https://doi.org/10.1145/3448016.3457256

[5] Atre M, Chaoji V, Zaki MJ, et al (2010) Matrix "bit" loaded: A scalable lightweight join query processor for RDF data. In: Proc. 19th International Conference on World Wide Web (WWW), pp 41–50, https://doi.org/10.1145/1772690.1772696

[6] Auer S, Bizer C, Kobilarov G, et al (2007) DBpedia: a nucleus for a Web of open data. In: Proc. 6th International Semantic Web (ISWC) Conference and 2nd Asian Semantic Web Conference (ASWC), pp 722–735, https://doi.org/10.1007/978-3-540-76298-0_52

[7] Barbay J, Claude F, Gagie T, et al (2014) Efficient fully-compressed sequence representations. Algorithmica 69(1):232–268. https://doi.org/10.1007/s00453-012-9726-3

[8] Berners-Lee T, Hendler J, Lassila O (2001) The semantic Web. Scientific American 284:34–43

[9] Bigerl A, Conrads F, Behning C, et al (2020) Tentris – A Tensor-Based Triple Store. In: Proc. 19th International Semantic Web Conference (ISWC), pp 56–73, https://doi.org/10.1007/978-3-030-62419-4_4

[10] Brisaboa NR, Ladra S, Navarro G (2014) Compact representation of Web graphs with extended functionality. Information Systems 39(1):152–174. https://doi.org/10.1016/j.is.2013.08.003

[11] Brisaboa NR, Cerdeira-Pena A, Fariña A, et al (2015) A compact RDF store using suffix arrays. In: Proc. 22nd International Symposium on String Processing and Information Retrieval (SPIRE), pp 103–115, https://doi.org/10.1007/978-3-319-23826-5_11

[12] Brisaboa NR, Cerdeira-Pena A, de Bernardo G, et al (2017) Compressed representation of dynamic binary relations with applications. Information Systems 69:106–123. https://doi.org/10.1016/j.is.2017.05.003

[13] Brisaboa NR, Cerdeira-Pena A, de Bernardo G, et al (2019) Improved compressed string dictionaries. In: Proc. 28th ACM International Conference on Information and Knowledge Management (CIKM), pp 29–38, https://doi.org/10.1145/3357384.3357972

[14] Brisaboa NR, Cerdeira-Pena A, de Bernardo G, et al (2020) Revisiting compact RDF stores based on k2-trees. In: Proc. 2020 Data Compression Conference (DCC), pp 123–132, https://doi.org/10.1109/DCC47342.2020.00020

[15] Chan HL, Hon WK, Lam TW, et al (2007) Compressed indexes for dynamic text collections. ACM Transactions on Algorithms 3(2):21–es. https://doi.org/10.1145/1240233.1240244

[16] Chantrapornchai C, Choksuchat C (2018) TripleID-Q: RDF query processing framework using GPU. IEEE Transactions on Parallel and Distributed Systems 29:2121–2135. https://doi.org/10.1109/TPDS.2018.2814567

[17] Clark D (1996) Compact PAT trees. PhD thesis, U. of Waterloo, Canada, https://doi.org/10.5555/287799

[18] Curé O, Blin, Guillaume, et al (2014) Waterfowl: A compact, self-indexed and inference-enabled immutable RDF store. In: Proc. 11th Extended Semantic Web Conference (ESWC), LNCS 8465, pp 302–316, https://doi.org/10.1007/978-3-319-07443-6_21

[19] Du J, Wang H, Ni Y, et al (2012) HadoopRDF: A scalable semantic data analytical engine. Intelligent Computing Theories and Applications 7390:633–641. https://doi.org/10.1007/978-3-642-31576-3_80

[20] Erling O (2012) Virtuoso, a hybrid RDBMS/graph column store. IEEE Data Eng Bull 35(1):3–8. URL http://dblp.uni-trier.de/db/journals/debu/debu35.html#Erling12

[21] Fariña A, Brisaboa NR, Navarro G, et al (2012) Word-based self-indexes for natural language text. ACM Transactions on Information Systems 30(1):1–34. https://doi.org/10.1145/2094072.2094073

[22] Fernández JD (2012) Binary RDF for scalable publishing, exchanging and consumption in the Web of Data. In: Proc. 21st International Conference on World Wide Web (WWW), pp 133–138, https://doi.org/10.1145/2187980.2187997

[23] Fernández JD, Martínez-Prieto MA, Gutiérrez C, et al (2013) Binary RDF representation for publication and exchange (HDT). Journal of Web Semantics 19:22–41. https://doi.org/10.1016/j.websem.2013.01.002

[24] Ferragina P, Manzini G (2005) Indexing compressed texts. Journal of the ACM 52(4):552–581. https://doi.org/10.1145/1082036.1082039

[25] Ferragina P, Venturini R (2010) The compressed permuterm index. ACM Transactions on Algorithms 7(1):article 10. https://doi.org/10.1145/1868237.1868248

[26] Ferragina P, Manzini G, Mäkinen V, et al (2007) Compressed representations of sequences and full-text indexes. ACM Transactions on Algorithms 3(2):article 20. https://doi.org/10.1145/1240233.1240243

[27] Gandon F, Schreiber G, (Editors) (2004) RDF 1.1 XML syntax, W3C recommendation. `https://www.w3.org/TR/rdf-syntax-grammar`

[28] González R, Grabowski S, Mäkinen V, et al (2005) Practical implementation of rank and select queries. In: Poster Proc. of 4th Workshop on Efficient and Experimental Algorithms (WEA), Greece, pp 27–38

[29] Grossi R, Vitter J (2000) Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In: Proc. 32nd ACM Symposium on Theory of Computing (STOC), pp 397–406, https://doi.org/10.1145/335305.335351

[30] Guo X, Gao H, Zou Z (2020) WISE: Workload-aware partitioning for RDF systems. Big Data Research 22:100,161. https://doi.org/10.1016/j.bdr.2020.100161

[31] Hammoud M, Rabbou D, Nouri R, et al (2015) DREAM: distributed RDF engine with adaptive query planner and minimal communication. Proc VLDB Endowment 8(6):654–665. https://doi.org/10.14778/

2735703.2735705

[32] Harbi R, Abdelaziz I, Kalnis P, et al (2016) Accelerating SPARQL queries by exploiting hash-based locality and adaptive partitioning. The VLDB Journal 25(3):355–380. https://doi.org/10.1007/s00778-016-0420-y

[33] Hogan A, Riveros C, Rojas C, et al (2019) A worst-case optimal join algorithm for sparql. In: Proc. 18th International Semantic Web Conference (ISWC), LNCS 11778, pp 258–275, https://doi.org/10.1007/978-3-030-30793-6_15

[34] Jacobson G (1989) Space-efficient static trees and graphs. In: 30th Annual Symposium on Foundations of Computer Science (FOCS), pp 549–554

[35] Jamour F, Abdelaziz I, Chen Y, et al (2019) Matrix algebra framework for portable, scalable and efficient query engines for RDF graphs. In: Proc. 14th EuroSys Conference 2019, article no.: 27, pp 1–15, https://doi.org/10.1145/3302424.3303962

[36] Jing Y, Jeong D, Baik DK (2009) SPARQL graph pattern rewriting for OWL-DL inference queries. Knowledge and Information Systems 20(2):243–262. https://doi.org/10.1007/s10115-008-0169-8

[37] Li W, Zhang B, Rao G, et al (2017) Hash tree indexing for fast SPARQL query in large scale rdf data management systems. In: Proc. 16th International Semantic Web Conference (ISWC) Posters & Demonstrations and Industry Tracks:, URL http://ceur-ws.org/Vol-1963/

[38] Mäkinen V, Navarro G (2008) Dynamic entropy-compressed sequences and full-text indexes. ACM Transactions on Algorithms 4(3):article 32. https://doi.org/10.1145/1367064.1367072

[39] Manber U, Myers G (1993) Suffix arrays: a new method for on-line string searches. SIAM Journal on Computing 22(5):935–948. https://doi.org/10.1137/0222058

[40] Manola F, Miller E, (Editors) (2004) RDF primer, W3C recommendation. `http://www.w3.org/TR/rdf-primer`

[41] Martínez-Prieto MA, Arias Gallego M, Fernández JD (2012) Exchange and consumption of huge RDF data. In: Proc. 9th Extended Semantic Web Conference (ESWC), pp 437–452, https://doi.org/10.1007/978-3-642-30284-8_36

[42] Martínez-Prieto MA, Fernández JD, Cánovas R (2012) Querying RDF dictionaries in compressed space. SIGAPP Applied Computing Review 12(2):64–77. https://doi.org/10.1145/2340416.2340422

[43] Martínez-Prieto MA, Brisaboa N, Cánovas R, et al (2016) Practical compressed string dictionaries. Information Systems 56(C):73–108. https://doi.org/10.1016/j.is.2015.08.008

[44] Munro JI (1996) Tables. In: Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS), LNCS 1180, pp 37–42, https://doi.org/10.1007/3-540-62034-6_35

[45] Munro JI, Nekrich Y, Vitter JS (2015) Dynamic data structures for document collections and graphs. In: Proc. 34th ACM Symposium on Principles of Database Systems (PODS), pp 277–289, https://doi.org/10.1145/2745754.2745778

[46] Navarro G, Mäkinen V (2007) Compressed full-text indexes. ACM Computing Surveys 39(1):article 2. https://doi.org/10.1145/1216370.1216372

[47] Nenov Y, Piro R, Motik B, et al (2015) Rdfox: A highly-scalable RDF store. In: Proc. 14th International Semantic Web Conference (ISWC), LNCS 9367, pp 3–20, https://doi.org/10.1007/978-3-319-25010-6_1

[48] Neumann T, Weikum G (2010) The RDF-3X engine for scalable management of RDF data. The VLDB Journal 19(1):91–113. https://doi.org/10.1007/s00778-009-0165-y

[49] Neumann T, Weikum G (2010) X-RDF-3X: Fast querying, high update rates, and consistency for RDF databases. Proc VLDB Endowment 3(1–2):256–263. https://doi.org/10.14778/1920841.1920877

[50] Ottaviano G, Venturini R (2014) Partitioned Elias-Fano indexes. In: Proc. 37th International ACM Conference on Research and Development in Information Retrieval (SIGIR), pp 273–282, https://doi.org/10.1145/2600428.2609615

[51] Perego R, Pibiri G, Venturini R (2021) Compressed indexes for fast search of semantic data. IEEE Transactions on Knowledge and Data Engineering 33(09):3187–3198. https://doi.org/10.1109/TKDE.2020.2966609

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

[53] Raman R, Raman V, Rao SS (2007) Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. ACM Transactions on Algorithms 3(4):article 8. https://doi.org/10.1145/1290672.1290680

[54] Sadakane K (2003) New text indexing functionalities of the compressed suffix arrays. Journal of Algorithms 48(2):294–313. https://doi.org/10.1016/S0196-6774(03)00087-7

[55] Sakr S, Al-Naymat G (2010) Relational processing of RDF queries: A survey. ACM SIGMOD Record 38(4):23–28. https://doi.org/10.1145/1815948.1815953

[56] Thompson BB, Personick M, Cutcher M (2014) The Bigdata® RDF graph database. In: Harth A, Hose K, Schenkel R (eds) Linked Data Management. Chapman and Hall/CRC, chap 8, p 1–46, https://doi.org/10.1201/b16859

[57] Weiss C, Karras P, Bernstein A (2008) Hexastore: Sextuple indexing for semantic web data management. Proc VLDB Endowment 1(1):1008–1019. https://doi.org/10.14778/1453856.1453965

[58] Yuan P, Liu P, Wu B, et al (2013) TripleBit: A fast and compact system for large scale RDF data. Proc VLDB Endowment 6(7):517–528. https://doi.org/10.14778/2536349.2536352

Appendix A: Summary of alternatives

Table 1 displays a brief summary of the main alternatives in the state of the art considered in this work. Note that all these tools have been tested in our experiments, except Waterfowl, for which there is no publicly available source code. The first group of solutions includes complex tools like Virtuoso or Blazegraph, that are widely used in academia and industry, as well as other solutions that provide a wide range of capabilities. The second group, where our proposal falls, includes tools designed for maximum compression of the RDF data, aiming at a full in-memory representation of the full dataset for efficient querying.

We omit solutions exploiting specific hardware, parallelism or designed for distributed environments, like WISE [30], Hadoop RDF [19], AdPart [32], TripleID-Q [16], and MAGiQ [35]. Additionally, we omit solutions like BMatrix [14], that is comparable to K2Triples+ but lacks join support, as well as other alternatives like Triplebit [58] that have been previously shown to be less efficient than the alternatives tested [9]. Finally, several existing solutions for which no publicly available code could be located, like HTStore [37] or RDFox [47], are also omitted.

Table 1 Summary of representative RDF stores and their main characteristics.

Tool	Year	Storage		Query capabilities			
		Format	Compr.	Joins	Updates	SPARQL	Endpoint
Virtuoso [20]	2012[1]	Tables	Low	Yes	Yes	Yes	Yes
Blazegraph [56]	2014[1]	Tables	Low	Yes	Yes	Yes	Yes
Tentris [9]	2020	Tensor	Low	Yes	No	Partial	Yes
MonetDB	2010	RDBMS	Low	Yes	Yes	No	No
RDF-3X [48]	2010	Multiple	Low	Yes	No[2]	Partial[2]	No[2]
HDT [23]	2013	CDS	High	No[3]	No	No[3]	No[3]
Waterfowl [18]	2014	CDS	High[4]	Yes[4]	No[4]	Yes[4]	No[4]
K2Triples [3]	2015	CDS	High	Yes	No	No	No
Perm. tries [51]	2020	CDS	High	No	No	No	No
RDFCSA	2022	CDS	High	Yes	No	No	No

[1] These are actively maintained and widely used tools. Years correspond to original publications.
[2] Later tools based on RDF-3X include updates, parallel processing and other capabilities [31, 49].
[3] Not considered in the C++ implementation. Supported in the (less efficient) Java library.
[4] Tool not publicly available. Data obtained from the cited publication.