REPOSITORIO UNIVERSIDADE CORUÑA

UNIVERSIDADE DA CORUÑA

This is an ACCEPTED VERSION of the following published document:

N. R. Brisaboa, A. Cerdeira-Pena, G. De Bernardo, y A. Fariña, «Revisiting Compact RDF Stores Based on k2-Trees», en 2020 Data Compression Conference (DCC), mar. 2020, pp. 123-132. doi: 10.1109/DCC47342.2020.00020.

Link to published version: https://doi.org/10.1109/DCC47342.2020.00020

General rights:

© 2020 IEEE. This version of the paper has been accepted for publication. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works. The final published paper is available online at: https:
https://doi.org/10.1109/DCC47342.2020.00020

Revisiting compact RDF stores based on k^2-trees

Nieves R. Brisaboa*, Ana Cerdeira-Pena*, Guillermo de Bernardo* and Antonio Fariña*

*Universidade da Coruña, Centro de investigación CITIC, Databases Lab.
A Coruña, Spain
{brisaboa,acerdeira,gdebernardo,fari}@udc.es

Abstract

We present a new compact representation to efficiently store and query large RDF datasets in main memory. Our proposal, called BMatrix, is based on the k^2-tree, a data structure devised to represent binary matrices in a compressed way, and aims at improving the results of previous state-of-the-art alternatives, especially in datasets with a relatively large number of predicates. We introduce our technique, together with some improvements on the basic k^2-tree that can be applied to our solution in order to boost compression. Experimental results in the flagship RDF dataset DBPedia show that our proposal achieves better compression than existing alternatives, while yielding competitive query times, particularly in the most frequent triple patterns and in queries with unbound predicate, in which we outperform existing solutions.

Introduction

The amount of valuable resources publicly available on the Web, in recent years, has increased to such an extent that new problems have arisen related to processing those resources. Getting insight into data and extracting knowledge from huge repositories of information has become a critical task. Based on the principles of the Semantic Web [1], the Web of Data has emerged as an effort to provide an environment of common access to the published data, by representing it through standard formats, so that it can be automatically reachable, and discovered.

The *Resource Description Framework* (RDF) [2] is a W3C recommendation to describe any resource in the form of triples *(subject, predicate, object)*. The popularity of RDF has led to the development of RDF stores, systems devoted to the storage of RDF data that also provide query support to access the stored information. The standard language to perform queries on RDF datasets is SPARQL [3], and basic graph patterns or *triple patterns* constitute its core. A triple pattern is a tuple (s, p, o), $s \in S$, $p \in P$ and $o \in O$, where each element can be set to a value or left unbound. For instance, $(s, p, ?)$ matches all the RDF triples that have subject s and predicate p.

RDF does not enforce an underlying storage format. Hence, a large number of works have been proposed in the last years to store and query RDF data, ranging from relational solutions [4] to native approaches [5–7]. As the popularity of RDF

Funded by: MICINN-AEI/FEDER-UE RTI2018-098309-B-C32, Xunta de Galicia/GAIN IN848D-2017-2350417; MICINN-AEI/FEDER-UE RTC-2017-5908-7; Xunta de Galicia/FEDER-UE ED431C 2017/58, ED431G/01, IN852A 2018/14; MINECO-AEI/FEDER-UE TIN2016-78011-C4-1-R, TIN2016-77158-C4-3-R, TIN2015-69951-R; EU H2020 MSCA RISE BIRDS: 690941

has been increasing, so has the size of RDF repositories. To handle these larger datasets, new approaches have been proposed: distributed stores [8, 9], and solutions based on compact data structures. For instance, k^2-triples [10] relies on vertical partitioning combined with a compact representation of binary matrices, called k^2-tree [11]; RDFCSA [12] is based on compressed suffix arrays [13].

In this paper we introduce a new representation based on k^2-trees, called BMatrix. Instead of resorting to vertical partitioning like k^2-triples, we aim at storing triples in a few data structures, in order to improve performance in RDF collections with a larger number of predicates. Particularly, BMatrix consists of two binary matrices, one related to triples subjects and the other to objects, and a small additional data structure; each matrix is stored using a k^2-tree. Experimental results show that our proposal beats k^2-triples, the most compressed representation in the state of the art up to now, in terms of space. We are also very competitive in query times, especially in the most used queries. Particularly, we are very efficient in queries with unbound predicate, where the k^2-triples needs additional indexes to be competitive in query times.

Related Work

In this section we present the basic data structures that are used in the k^2-tree, as well as the k^2-tree itself, since they are necessary for understanding our proposal.

A *bit sequence* or *bitmap* is a sequence of n bits, $B[1, n]$, that supports three basic operations: $rank_c(B, i)$ counts the number of occurrences of bit c in B up to position i; $select_c(B, j)$ returns the position in B of the j-th bit set to c; and $access(B, i)$ gets the bit value at $B[i]$. These operations can be answered in constant time using $o(n)$ bits in addition to the bitmap [14]. Compressed representations [15] can answer the same operations while compressing the bitmap. In this work we use a practical implementation [16] that is based on single-level sampling. The default setup adds a 5% of space overhead and provides efficient query times.

Directly addressable codes [17], or DACs, is a technique that provides direct access to sequences of variable-length codes, where each codeword can be regarded as a sequence of chunks of b bits each, for any fixed b. DACs works by reorganizing these chunks in several arrays, L_i, and using additional bitmaps, B_i, to mark for each entry in L_i whether the corresponding word has a next chunk in L_{i+1} or not. In that way, entries can be decompressed by accessing the first chunk directly and then using $rank_1$ operations on the B_is to locate the corresponding position of the next chunk.

The k^2*-tree* [11] is a compact representation of sparse binary matrices originally devised for Web graphs. Given an $n \times n$ matrix, it is represented as a k^2-ary tree, for a fixed k. The root of the tree represents the complete matrix. The matrix is subdivided in k^2 submatrices of equal size. These submatrices are read in a left-to-right and top-to-bottom order, and for each of them a child node is appended to the tree root. Each node is marked with a single bit: 1 if the submatrix contains at least a 1, and 0 otherwise. The decomposition process continues recursively for each 1-bit, until we reach the cells of the original matrix. The conceptual k^2-tree is traversed levelwise, and its bits are stored using just two bitmaps: T stores the bits from all

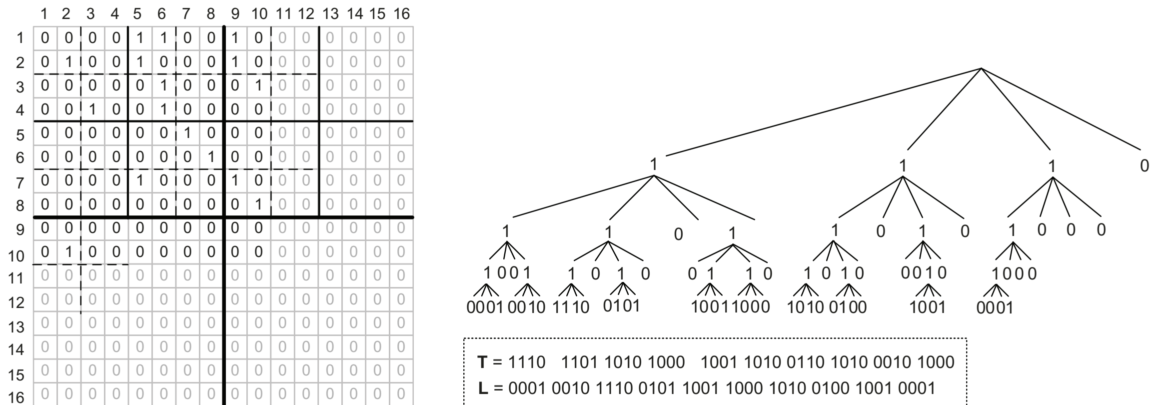

Figure 1: Example of binary matrix and its associated k^2-tree.

the levels except the last one; L stores the bits from the last level. Figure 1 shows a conceptual matrix, its k^2-tree for $k = 2$, and the corresponding T and L bitmaps.

Single cell, row/column, and bi-dimensional range queries can be answered by means of traversals of the conceptual tree, starting at the root and traversing only the submatrices intersecting the queried region. Top-down traversal of the tree can be replicated in the bitmaps T and L using the following property: given an internal node at position p ($T[p] = 1$), its k^2 children are consecutive and start at position $p' = rank_1(T, p) \times k^2$ in T:L (i.e. the concatenation of T and L).

Some modifications to the basic k^2-tree have been presented by the original authors. The most relevant one is the statistical compression of the lower levels of the conceptual tree to exploit small-scale regularities in the binary matrix. This is achieved by building a matrix vocabulary and representing each matrix by its identifier; the sequence of encoded matrices is then stored using DACs.

The k^2-triples [10] is a solution based on k^2-trees to represent RDF datasets. It applies vertical partitioning to the RDF dataset, creating $|P|$ binary matrices that correspond to the pairs (s, o) associated with each predicate. Each of those matrices is represented with a k^2-tree. In this solution, triple patterns can be easily translated into k^2-tree operations. For instance, an (s, p, o) query is solved by checking cell (s, o) in the k^2-tree associated to p. The query $(s, ?, ?)$ is translated into $|P|$ queries asking for all the elements in row s in each k^2-tree. Notice that, whenever the predicate of a query is unbound, the k^2-triples must query all the k^2-trees, which may be costly in datasets with a large number of predicates. An enhanced variant of the data structure, called k^2-triples$^+$, uses additional indexes to cope with this problem.

Our proposal: BMatrix

As explained in previous sections, techniques based on k^2-trees have shown good compression capabilities over RDF data. However, the vertical partitioning used in k^2-triples makes the structure slow to answer queries with unbound predicates. Using additional indexes partially solves the problem, but leads to significantly larger space

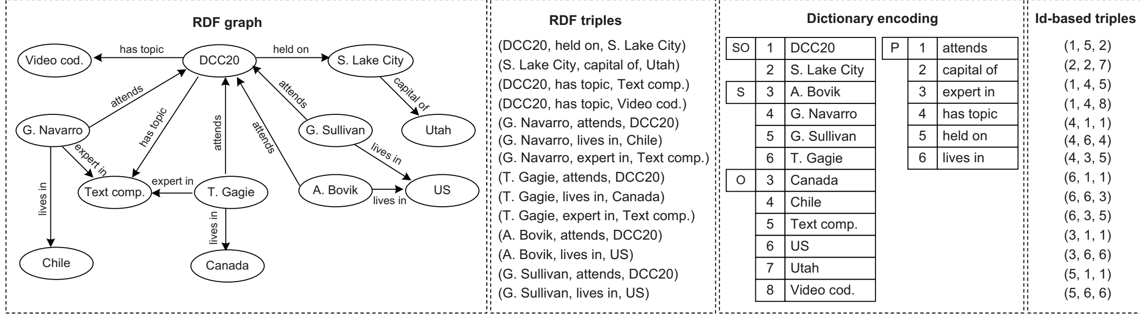

Figure 2: Example of RDF graph and its corresponding dictionary encoding.

requirements. Our proposal, that we call BMatrix, also uses the k^2-tree as underlying data structure, but we follow a different approach for organizing the RDF data. Particularly, our goal is to use fewer structures, and keep all the predicates together.

Our representation is designed to store RDF triples encoded as integer identifiers. Therefore, it requires a dictionary to encode/decode the original strings into integer ids. We follow the same scheme for dictionary encoding used by state-of-the art solutions based on compact data structures like k^2-triples and RDFCSA. Figure 2 shows an example of RDF graph and the corresponding dictionary encoding. Strings are divided in four categories: subject–objects (strings that appear as both subjects and objects), subjects-only, objects-only and predicates. Consecutive ids are assigned to each unique string in each category (notice that subject-only and object-only entries start numbering after the last subject–object).

After dictionary encoding, we have a collection T of n triples $t_i = (s_i, p_i, o_i)$, where each s_i, p_i and o_i is an integer. triples with the same predicate are grouped together. We use p, o, s order since it leads to better compression in practice, but any other ordering that groups triples with the same p could be used. After sorting, we build two binary matrices ST and OT. ST has $|S|$ rows and $n = |T|$ columns, and a cell (r, c) in ST is set to 1 iff $s_c = r$. OT is similar, but has $|O|$ rows, and a cell (r, c) in OT is set to 1 iff $o_c = r$. Notice that only a single 1 can appear in each column of ST and OT. Figure 3 shows the matrices generated for the RDF dataset of Figure 2. Note that the grayed out portions of the matrices do not belong to the conceptual representation. However, each matrix will then be stored using a k^2-tree, that conceptually expands the matrix to the next power of k.

In order to recover the original triples, we also need to store an auxiliary structure to know the column ranges corresponding to each predicate. We can use any bitmap BP of length n, storing a 1 for the positions where the predicate changes; in this representation, the predicate of a triple t_i can be computed as $rank_1(BP, i)$, and the starting position of a predicate p_i as $select_1(BP, i)$. We use a custom representation supporting those operations as follows: we set an array AP of size $|P|$ storing the initial position of each predicate. Additionally, we select a sampling period d and build an array $rankP$, that stores the predicate that contains triple $d \cdot i$ for $i \in$

Figure 3: BMatrix conceptual representation.

$[1..n/d]$. In this representation, $select_1(BP, i)$ is computed as $AP[i]$. To answer $rank_1(BP, i)$, we use $rankP$ to identify the range of predicates that could contain that triple $[rankP[i/d], rankP[i/d + 1]]$, and binary search in AP for the rightmost entry that is not greater than i. Assuming that $|P| << n$ and for relatively large d, the space required by this structure is much smaller than n.

Query operations

In this section we describe the implementation of triple pattern queries in our representation. Essentially, we reduce triple patterns to operations on k^2-trees ST and OT and $rank/select$ operations on bitmap BP.

(s, p, o) queries just require checking that the triple pattern exists in the collection. First we find all the triples that have subject s and predicate p. To do this, we compute the range of columns corresponding to p as $[select_1(BP, p), select_1(BP, p + 1) - 1]$. Then, we search for all the ones in SP in row s and in the given range of columns (this operation is implemented in a k^2-tree like a simple row search by adding filters at each step that restrict search to branches inside the column range). For each result t_i found in ST, we perform a cell retrieval query in OT for cell (o, i), in order to check if the triple had o as its object. We return immediately when a single result is found. Notice that we could also perform the operation starting in OT and checking in ST, but our results suggest that this alternative is slower due to the large number of intermediate results generated in queries with very common objects.

$(s, p, ?)$ queries start, like the previous ones, by finding in ST all the triples with subject s and predicate p. Then, for each result obtained in ST, we need to perform a column query in OT to obtain the object for that triple. Notice that, since our matrices have a single 1 per column, column queries can return immediately when they find a single result. $(?, p, o)$ queries are symmetrical to $(s, p, ?)$, starting queries in OT and then extracting results in ST.

$(s,?,o)$ queries are implemented by first finding all the triples for object o, with a row query in OT. Then, we continue depending on the number of partial results: if the number of intermediate results is small, we simply check each result t_i in ST with a cell query for (s,i); if the number of intermediate results is large, we perform a second row query, now in ST, to get all the triples for subject s, and intersect both lists to obtain the final result (the intersection is very efficient since both lists are already sorted). In practice, the threshold value $t_{merge-unsorted}$ can be a small value (e.g., 10), since column queries in k^2-trees are roughly an order of magnitude slower than cell retrieval queries.

$(s,?,?)$ queries start again by finding all the triples for subject s with a row query in ST. For each partial result t_i, we run a column query in OT to get the corresponding object. Additionally, we must compute the predicate for each tuple as $rank(BP,i)$. $(?,?,o)$ queries are symmetrical to $(s,?,?)$, performing the row query in OT and the column queries in ST.

$(?,p,?)$ queries involve finding all the cells for a given predicate. We start by obtaining all the subjects for those triples: we compute the column range for predicate p $([select_1(BP,p), select_1(BP,p+1)-1])$ and perform a range query in ST limiting columns to the given range. This yields a list of (s,i) pairs that will be the results of our query. In order to obtain the corresponding objects, we again check the number of partial results: if it is small, we simply perform a column query in OT per result; if it is larger, we perform a second range query in OT, to get a list of (o,i) pairs, and intersect the resulting lists to obtain the final result (in this case, lists are not sorted by column, so we sort them before merging). We use a different threshold $t_{merge-unsorted}$, but again a relatively small threshold can be used in practice to guarantee the best overall performance and more stable times in queries with many intermediate results.

Space improvements

Taking advantage of our setup, we can reduce significantly the size of the vocabulary with simple representations. Consider a matrix vocabulary with m matrices of size $k_L \times k_L$. We view the same vocabulary as a set of mk_L columns. We build a bitmap C, of size mk_L, so that $C[i]=1$ if the corresponding column has a 1. Then, we use a separate array R to store the rows containing the ones, requiring $\log_2 k_L$ bits per entry (k_L is assumed to be a power of two).

We can simply create an array R with mk_L entries, and set to 0 columns without a value. This means that we can store the complete vocabulary using $mk_L(1+\log_2 k_L)$ bits. For any entry e in the vocabulary, we can recover the value at (r,c) in its submatrix by checking $C[ek_L+c]$); if it is 0, the value is 0; if it is 1, we check if $R[ek_L+c]$ is equal to r. This solution provides minimum query overhead, replacing the bit access of the plain vocabulary with a few array accesses and checks.

A more elaborate scheme can be built storing an entry in R only for columns that have a 1. In this variant, to access position (r,c) in matrix e, we first check the bit $C[ek_L+c]$, like in the previous alternative; if it is 0, the value is 0; if it is 1, we must access R at position $p' = rank_1(C, ek_L+c)$; and check if $R[p']$ is equal to r. This variant can save a significant amount of space compared to the previous one, but has a significant overhead in practice due to the complexity of the rank operation.

Experimental Evaluation

Our proposal is designed to work well in datasets with a relatively large number of predicates, where other alternatives like k^2-triples require additional space to efficiently answer queries. We evaluated the compression and query performance of our solution using the *DBPedia* dataset[1], a widely used, large and heterogeneous RDF collection. The original size of DBPedia, considering triples storing string values, is around 34 GB, and it contains 232M triples. After applying dictionary compression to the dataset, the collection of triple identifiers can be stored in 2.6GB (i.e., 3 integers per triple). The dataset contains 18.4M different subjects, 39,672 predicates and 65.2M different objects. We use an existing testbed [2] that includes 500 queries of each triple pattern. For each pattern, we determine a minimum number of repetitions necessary to obtain consistent times and measure the average query times per result.

We compare our representation with two state-of-the-art approaches based on compact data structures: k^2-triples and its extension k^2-triples$^+$, and the RDFCSA. Both techniques have been shown to overcome alternative solutions in space, and provide very efficient query times for most triple pattern queries. For RDFCSA we use the default configuration, and test sampling values $t_\Psi \in \{16, 32, 64, 256\}$; among the query implementations provided by the authors, we show results in our experiments for the binary search, that can be applied to all triple patterns and is the most consistent in query times. For k^2-triples, we use a hybrid representation with $k = 4$ in the first 5 levels of decomposition and $k = 2$ in the remaining levels. The bitmaps use the default rank structure, requiring an extra 5% space. The lower levels of the tree are compressed using a matrix vocabulary of 8×8. In queries that have unbound predicate, we show the tradeoff obtained by the basic version (smaller, slower) and the k^2-triples$^+$ version with additional indexes (larger and faster).

To provide the fairest possible comparison with k^2-triples, our k^2-trees use the same exact configuration, the only difference being the vocabulary representation: we use the simplest representation of the vocabulary proposed in the previous section, in order to obtain some space savings with minimal overhead. Additionally, we show results for our representation with denser sampling in the k^2-trees bitmaps, requiring 12.5% extra space; this leads to a larger solution with faster query times. This is used just to outline the level of tradeoff that can be obtained tuning this parameter in k^2-trees; notice that a similar tradeoff can be achieved in k^2-triples.

We run our experiments on an Intel Xeon E5-2470@2.3GHz (8 cores) CPU, with 64GB RAM. The operating system was Debian 9.8 (kernel 4.9.0-8-amd64). Our code is implemented in C and compiled with gcc 6.3.0 with the -O9 optimization flag.

Results

We measure the compression and query efficiency of our proposal on the seven triple patterns that compose the basis of SPARQL queries. We divide our experimentation in two main groups of patterns: the four plots at the top of Figure 4 display results

[1] `http://downloads.dbpedia.org/3.5.1/`

[2] Available at `http://dataweb.infor.uva.es/queries-k2triples.tgz`, provided by the authors of k^2-triples.

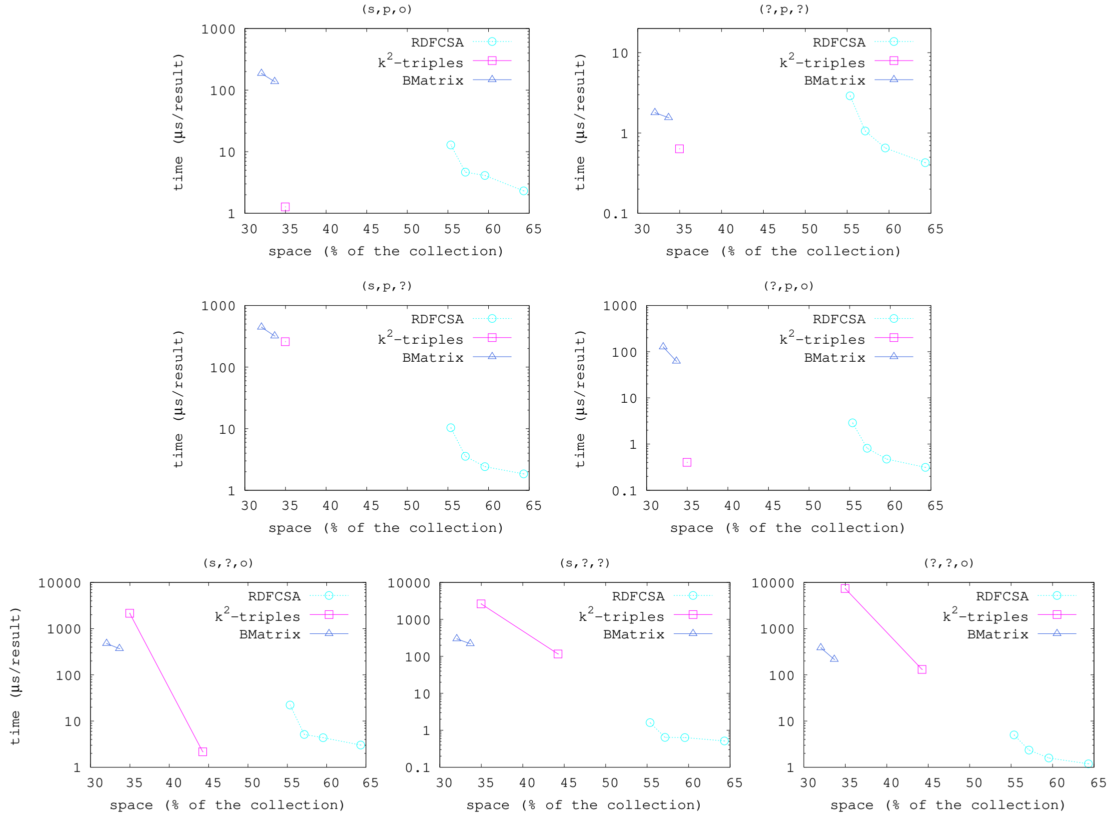

Figure 4: Space/time tradeoff for triple patterns. Space measured as percentage of the input size. Query times in μs per result.

for triple patterns with fixed predicate, while the three at the bottom display results for patterns with unbound predicate.

Results show that BMatrix significantly improves the compression of k^2-triples. With sampling similar to k^2-trees, we require 10% less space. Even using the denser sampling, we are still smaller than k^2-triples. Furthermore, when k^2-triples needs to use additional indexes it becomes much larger than BMatrix. RDFCSA is significantly larger than any of the other alternatives.

As shown in Figure 4 (top), query times for patterns with fixed predicate in BMatrix are quite consistent, but comparison results are significantly different depending on the triple pattern: in (s, p, o) queries, k^2-triples just needs to run a cell query in a k^2-tree, whereas our algorithm is more complex; this leads to our implementation becoming significantly slower. In $(?, p, ?)$ queries, we are still slower than k^2-triples but much closer, since range operations are required in both cases; query times are also comparable to those of RDFCSA using about half their space. In $(s, p, ?)$ queries we are comparable to k^2-triples, but RDFCSA is very efficient and provides an interesting tradeoff. In $(?, p, o)$ queries BMatrix is again significantly slower than k^2-triples, that is clearly the best alternative. Notice that this query is essentially equivalent to the previous one in complexity, both in k^2-triples and BMatrix, but k^2-triples is much more efficient in $(?, p, o)$, due to the usually much larger number of results per

query in $(?, p, o)$ queries. In BMatrix, we have to extract the object individually for each result, so having a few queries that yield many results has a significant impact on our performance.

For triple patterns with unbound predicate, Figure 4 (bottom) displays two points as space/time tradeoff for k^2-triples; these correspond to the basic implementation and the k^2-triples$^+$ variant with additional indexes. Results show that BMatrix significantly improves the query times of the basic k^2-triples implementation. In $(s, ?, o)$ queries, k^2-triples is still the fastest technique when using extra indexes, but to do this it requires 40% more space than BMatrix. In $(s, ?, ?)$ and $(?, ?, o)$ queries, we are an order of magnitude faster than k^2-triples without extra indexes, and even comparable in query times to the k^2-triples version that uses extra 40% space. RDFCSA is faster than BMatrix, but almost twice as large, so our proposal is still the best option when memory usage is an issue.

Taking into account the different comparison results obtained, BMatrix provides a very reasonable space/time tradeoff depending on the types of queries to be executed: when a large percentage of triple patterns with unbound predicate are expected, BMatrix clearly overcomes k^2-triples and provides a very compact alternative to RDFCSA. Additionally, our experiments show that BMatrix is more competitive in the triple patterns that are more frequently used: a previous analysis on the DBPedia dataset [18] has shown that 90% of the triple patterns used in SPARQL queries over DBPedia are $(s, p, ?)$, where BMatrix is competitive with k^2-triples, and $(s, ?, ?)$, where BMatrix is either much faster or much smaller than the alternatives.

Conclusions and Future Work

We have introduced BMatrix, a compact representation of RDF datasets based on k^2-trees. It aims mainly at improving the performance of previous solutions in datasets with a relatively large number of predicates, where the vertical partitioning strategy leads to poor query times in patterns with unbound predicate. As a side result, we also propose some simple improvements on the k^2-tree data structure that have stand-alone interest and could be applied to other domains.

We experimentally evaluate our proposal on DBPedia, a widely used RDF dataset containing around 40,000 predicates. We compare our proposal with k^2-triples and RDFCSA. Our results show that BMatrix achieves better compression, being 10% smaller than k^2-triples and 40–50% smaller than RDFCSA. BMatrix is also competitive in query times in the most frequent query patterns. In query patterns with unbound predicate, BMatrix is faster than the basic k^2-triples. For $(s, ?, ?)$ and $(?, ?, o)$ queries, we obtain query times comparable to those of the most efficient k^2-triples version with extra indexes, that uses 40% more space than our proposal.

Currently, BMatrix supports all basic triple patterns. We plan to extend our evaluation to multi-pattern join queries, that can be supported by merging or chained evaluation of individual triple patterns, as in state-of-the-art alternatives. Synchronized traversal of multiple k^2-trees, used in k^2-triples, can also be applied to our solution in order to improve query times. Finally, we believe that new tradeoffs can be obtained in solutions based on k^2-trees to speed up specific queries. BMatrix aims

at boosting queries with unbound predicate, that are relevant for many application domains, but some other arrangements could benefit other application scenarios.

References

[1] T. Berners-Lee, J. Hendler, and O. Lassila, "The semantic web," *Scientific American Magazine*, 2001.

[2] F. Manola and E. Miller, "RDF Primer," 2004.

[3] S. Harris, E. Prud'hommeaux, and A. Seaborne, "SPARQL query language for RDF, W3C recommendation," `http://www.w3.org/TR/rdf-sparql-query`, 2008.

[4] S. Sakr and G. Al-Naymat, "Relational processing of RDF queries: A survey," *ACM SIGMOD Record*, vol. 38, no. 4, pp. 23–28, 2010.

[5] T. Neumann and G. Weikum, "The RDF-3X engine for scalable management of RDF data," *The VLDB Journal*, vol. 19, no. 1, pp. 91–113, 2010.

[6] "MonetDB," `http://www.monetdb.org`, 2013.

[7] O. Curé, G. Blin, D. Revuz, and D. Faye, "Waterfowl: A compact, self-indexed and inference-enabled immutable RDF store," in *Proc. European Semantic Web Conference*, 2014, pp. 302–316.

[8] J. Du, H. Wang, Y. Ni, and Y. Yu, "HadoopRDF: A scalable semantic data analytical engine," *Intell. Computing Theories and Applications*, vol. 7390, pp. 633–641, 2012.

[9] M. Hammoud, D. Rabbou, R. Nouri, S. Beheshti, and S. Sakr, "Dream: distributed rdf engine with adaptive query planner and minimal communication," *Proc. of the VLDB Endowment*, vol. 8, no. 6, pp. 654–665, 2015.

[10] S. Álvarez-García, N. Brisaboa, J. Fernández, M. Martínez-Prieto, and G. Navarro, "Compressed vertical partitioning for efficient rdf management," *Knowledge and Information Systems*, vol. 44, no. 2, pp. 439–474, 2015.

[11] N. R. Brisaboa, S. Ladra, and G. Navarro, "Compact representation of web graphs with extended functionality," *Information Systems*, vol. 39, pp. 152–174, 2014.

[12] N. R. Brisaboa, A. Cerdeira-Pena, A. Fariña, and G. Navarro, "A compact rdf store using suffix arrays," in *Proc. String Processing and Information Retrieval*, 2015, pp. 103–115.

[13] K. Sadakane, "New text indexing functionalities of the compressed suffix arrays," *Journal of Algorithms*, vol. 48, no. 2, pp. 294–313, 2003.

[14] G. Navarro, *Compact Data Structures: A Practical Approach*, Cambridge University Press, 2016.

[15] D. Okanohara and K. Sadakane, "Practical entropy-compressed rank/select dictionary," in *Proc. Meeting on Algorithm Engineering & Experiments*, 2007, pp. 60–70.

[16] R. González, Sz. Grabowski, V. Mäkinen, and G. Navarro, "Practical implementation of rank and select queries," in *Proc. Workshop on Efficient and Experimental Algorithms*, 2005, pp. 27–38.

[17] N. R. Brisaboa, S. Ladra, and G. Navarro, "DACs: Bringing direct access to variable-length codes," *Inf. Processing and Management*, vol. 49, no. 1, pp. 392–404, 2013.

[18] M. Arias, J. D. Fernández, M. A. Martínez-Prieto, and P. de la Fuente, "An empirical study of real-world SPARQL queries," in *Proc. Workshop on Usage Analysis and the Web of Data*, 2011.