Interleaved K^2-tree: Indexing and Navigating Ternary Relations *

Sandra Álvarez-García[a] Nieves R. Brisaboa[a] Guillermo de Bernardo[a]
Gonzalo Navarro[b]

[a]*Database Laboratory, University of A Coruña,Spain*
{*brisaboa,sandra.alvarez,gdebernardo*}*@udc.es*

[b]*Department of Computer Science, University of Chile*
gnavarro@dcc.uchile.cl

Abstract: We propose a new compressed and self-indexed data structure that we call Interleaved K^2-tree (IK^2-tree), designed to compactly represent and efficiently query general ternary relations. The IK^2-tree is an evolution of the K^2-tree [8], initially designed to represent Web graphs but later used to represent general binary relations. The IK^2-tree represents at the same time the three dimensions in the ternary relation and provides indexing capabilities over the three of them, but it also offers other interesting features to improve some types of queries over one of the three dimensions, the dimension used in the nodes of the trees instead of in the organization of the branches.

1 Introduction

Graphs are a natural way to represent complex data. They can be seen as binary relations, and have been intensively studied [9, 18] especially in specific domains such as Web graphs [5, 3]. The K^2-tree is a compressed and self-indexed structure initially designed for Web graphs [8, 10] and later used in other domains [2, 12] for the compact representation and efficient querying of binary relations.

Ternary relations did not receive so much attention, but they also appear everywhere. For example any dynamic binary relation has a temporal dimension and therefore becomes a ternary relation (we call them temporal graphs in this paper). Images and raster data can be seen as the relations among rows, columns and values. Linked Data uses RDF representations, where a set of triples (S,P,O) represents the relations among subjects, objects and predicates. However, little attention has been paid to the efficient representation and management of general ternary relations [4]; most of the approaches are domain oriented such as *Geotiff* [16] for the representation of raster data or *RDF-3X* [15] for RDF.

A usual strategy for the representation of ternary relations is to create a *vertical partitioning* [1] which reduces the problem to efficiently store and query several binary relations, one for each value of the partitioning variable.

In this paper we propose the Interleaved K^2-tree (IK^2-tree), a compressed and self-indexed structure to represent and query general ternary relations that gathers in a single tree the three dimensions, providing indexing capabilities over all of them. We also experimentally evaluate the IK^2-tree on RDF and temporal graphs.

*SAG, NB and GdB were founded by MICINN (PGE and FEDER) grants TIN2009-14560-C03-02, TIN2010-21246-C02-01 and Xunta de Galicia (co-funded with FEDER) ref. GRC2013/053. GN was partially funded by Millennium Nucleus Information and Coordination in Networks ICM/FIC P10-024F

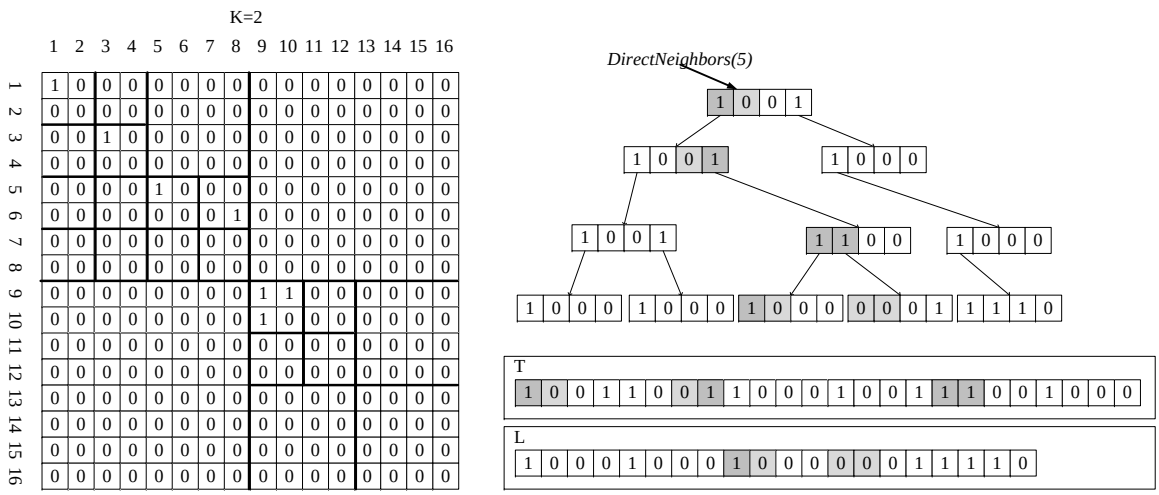

Figure 1: An example of a binary relation represented with a K^2-tree

2 Previous Work: The K^2-Tree

The K^2-tree [8] is a compact data structure to represent binary relations, conceptually represented by a binary adjacency matrix M, where $M[i, j]$ is 1 if element i is related with element j and 0 otherwise. It was originally designed to represent Web graphs. The K^2-tree takes advantage of the sparsity of the matrix (large areas of *zeros*) and the clustering (proximity) of the *ones*. It achieves very low space (less than 5 bits per link) over Web graphs, allowing large graphs to fit in main memory. It also supports efficient navigation over the compressed graph [8], efficiently answering direct and reverse neighbor queries, individual cell and range queries.

The K^2-tree conceptually subdivides the adjacency matrix into K^2 submatrices of equal size. Each of the K^2 submatrices is represented with one bit in the first level of the tree, following a left to right and top to bottom order. The bit that represents each submatrix will be 1 if the submatrix contains at least one cell with value 1. Otherwise, if it is an empty area, the bit will be a 0. The next level of the tree is created by expanding the 1 elements of the current level, subdividing the submatrix they represent. In this way, K^2 children are created in the next level to represent the new subdivisions. This method continues recursively until the subdivision reaches the cell-level. Fig. 1 shows an example of this tree for $K = 2$. The first 1 of the first level (root) means the upper-left 8×8 submatrix has at least a cell with value 1. The second bit of the root is a 0, which shows that the upper-right submatrix does not contain any relation between nodes, and so on.

The K^2-tree is stored with only two bitmaps: T for the intermediate levels of the K^2-tree, following a levelwise traversal, and L for the bits of the last level (Fig. 1).

Retrieving direct or reverse neighbors requires obtaining the cells with value 1 for a given row or column in the adjacency matrix. Both operations are symmetric. They are solved in the K^2-tree by a top-down traversal over the tree for the two appropriate branches of each node. The example shows the bits of the tree traversed in order to obtain the direct neighbors of row 5 (i.e., the *ones* in the 5th matrix row).

This navigation over the K^2-tree is efficiently performed over the bitmaps T and L. Given a 1 at position x in T, the children of x are K^2 bits placed in $T : L$ starting at position $rank_1(T, x) \times K^2$, where $rank_1$ counts the number of ones in $T[1..x]$. Rank operations are performed in constant time by using an additional *rank structure*, created over the bitmap T, that requires sublinear space in addition to T [14].

In the worst case the space in bits is $K^2 e(\log_{K^2} \frac{n^2}{e} + O(1))$, where n is the number of nodes and e the number of *ones*. Retrieving direct or reverse neighbors in the worst case is $O(n)$ time, although the time is much better in practice.

The implementation of the K^2-tree allows using different K values depending on the level of the tree (*hybrid* approach) or compressing the last levels of the conceptual tree using a vocabulary of submatrices encoded with a statistically compressor. *Direct Access Codes* [6] are used to provide direct access to each code. A dynamic variant of the K^2-tree that combines good compression ratios with fast query and update times has been proposed [7]. Other variants compress efficiently not only large regions of zeros but also regions of ones [11].

3 Interleaved K^2-tree

We define a ternary relation as a set of triples $T = \{(x_i, y_j, z_k)\} \subseteq X \times Y \times Z$. An approach to represent a ternary relation uses vertical partitioning to transform T into $|Y|$ binary relations T_j, one for each different value $y_j \in Y$. Each T_j will contain the pairs (x_i, z_k) that are related with y_j. The dimension Y is called *partitioning variable*. Each of the binary relations can be stored in a structure designed for managing binary relations, such as the K^2-tree. In this way, the set of K^2-trees composes a complete system, that can efficiently answer the relevant queries.

In this work we propose the Interleaved K^2-tree (IK^2-tree), that is an evolution of this vertical partitioning approach. It represents ternary relations by gathering $|Y|$ K^2-trees in the same tree, providing indexing capabilities also on the variable Y.

3.1 Data structure

Given the decomposition of a ternary relation into $|Y|$ adjacency matrices, the IK^2-tree represents all those matrices simultaneously. Each node of the IK^2-tree represents, just as in the K^2-tree, a submatrix containing the relations between the rows (values of X) and the columns (values of Z), but instead of using only one bit to know whether the matrix is empty or not, it contains a variable number (from 1 to $|Y|$) of bits. Root nodes contain $|Y|$ bits, one per $y_j \in Y$. For a node N_i with m *ones*, each of its children nodes will contain m bits, one per submatrix that contains a *one* in N_i.

Fig. 2 shows an example IK^2-tree, where the partitioning variable Y can take three distinct values. The figure shows three adjacency matrices, one per value of Y. The first node of the first level of the IK^2-tree (N_0) has a *zero* at the first position, because the top-left submatrix of the adjacency matrix y_0 does not contain any *one*.

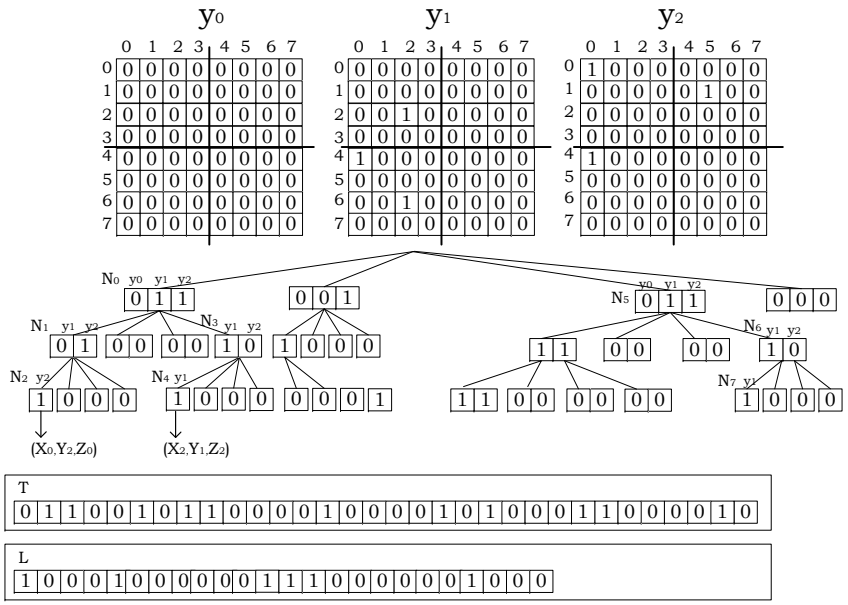

Figure 2: An example of a ternary relation represented with the IK^2-tree

The second bit of N_0 is a *one* because the adjacency matrix corresponding to y_1 has a *one* in the top-left submatrix. Thus, we have a first level of the tree with $|Y| \times K^2$ bits representing the submatrices of the first division of the adjacency matrix.

Each node in an intermediate level of the IK^2-tree will have K^2 children nodes in the next level (like the original K^2-tree), but the number of bits of the children is given by the number of *ones* of the parent, that is, the number of *active* Y values in the parent node. For instance, node N_0 in Fig. 2 has K^2 children of 2 bits each, since only the values y_1 and y_2 are *ones*. If all the bits of a node are *zero*, it has no children. Otherwise the construction continues recursively down to the leaves, where each bit corresponds exactly to a cell of one of the possible adjacency matrices in the vertical partition. The resulting tree is stored, using the same scheme of the original K^2-tree, in two bitmaps: T for the intermediate levels and L for the last level.

Although the number of bits per node changes, a *one* in a level i produces always K^2 bits in level $i+1$, even if those bits are not consecutive. As a result, the navigation over the tree is as in the original K^2-tree. For a node starting at position x and having at least a *one* in its bitmap, its first child starts at position $rank_1(T, x-1) \times K^2 + adjust$, in $T : L$, where $adjust = (|Y|) \times K^2$ is the number of bits at the first level.

Notice that the IK^2-tree is an aggregation and reorganization of the same bits used by the set of K^2-trees representing the $|Y|$ relations. While using similar space, this reorganization supports efficient queries even when the variable Y is unbounded.

Although the three dimensions are represented in the same tree, X and Z dimensions have a more efficient indexing than the dimension Y, since they directly support the navigation over the tree by pruning the branches according to the queried values for X and Z. Pruning the tree by the Y dimension implies more complex operations, like having into account the active values for each node. Therefore, in general, the most convenient partitioning variable will be the smallest one, although this criterion can change depending on the typical queries of the domain.

3.2 Basic navigation

3.2.1 Query patterns with bounded partitioning variable

Some queries specify a fixed value that the partitioning variable has to take. The query patterns that belong to this group are (x_i, y_j, z_k), $(*, y_j, z_k)$, $(x_i, y_j, *)$, and $(*, y_j, *)$, where $*$ means the variables are unbounded, i.e., they can take any value.

Fig. 2 shows the nodes involved to solve the query (x_6, y_1, z_0). In the root of the tree, we explore the third node (N_5) because (x_6, z_0) is in the bottom-left submatrix. Since all nodes in the root have $|Y|$ bits we know that N_5 is in positions [6..8] in T. The second bit of N_5, corresponding to y_1, is a *one*, therefore we go down to the children of N_5. They start at position $rank_1(T, 5) \times K^2 + adjust = 24$ in T and have 2 bits each (the number of *ones* in N_5). The second bit in N_5 is the first *one* in this node, so the bit corresponding to y_1 will be the first in each child. To find the values for (x_6, z_0) we go to the fourth child (N_6) and, because we want to check y_1, we check its first bit. Following the same idea, we check the first bit of N_7, that is a *one*, therefore the triple (x_6, y_1, z_0) exists. The other three query patterns are solved in a similar way. The process is exactly as it would be in the individual K^2-tree of y_j, using X and Z to drive the traversal, but also tracing the bit corresponding to the desired value of variable Y in the bitmaps of the nodes.

3.2.2 Queries with unbounded partitioning variable

The IK^2-tree allows the efficient execution of queries with the variable Y unbounded. The patterns of those queries are $(x_i, *, z_k)$, $(*, *, z_k)$, $(x_i, *, *)$, and $(*, *, *)$.

The operation starts from the top of the tree. For each chosen node N_j in the root level (depending on the variables X and Z), we store a list A_j with the active values for variable Y in node N_j. In the next levels, the active list is built, from the parent list, by storing only its elements having value 1 in the current node.

For instance, using again Fig. 2, if we want to answer the query $(x_2, *, z_2)$ we check the first node of the tree, N_0. The list of active values for N_0 is $A_0 = \{y_1, y_2\}$. Therefore when we go to the children of N_0, we know that the two bits of each node correspond to y_1 and y_2 respectively. The child that maps $(x_2, *, z_2)$ is N_3, so we create a new list of active values $A_3 = \{y_1\}$ (because node N_3 has a 0 in the position corresponding to y_2). Then we look for the children of N_3. Each child contains only one bit, which corresponds to the only active value in A_3 (that was y_1). The child corresponding to the cell (x_2, z_2) is the first one, and it has a value 1, therefore we can answer the query with the triple (x_2, y_1, z_2). The other queries of this group are implemented in the same way by maintaining the correspondence of each bit position with the values of Y, that is, the active values of Y for each node.

4 Using the IK^2-tree to Represent RDF Databases

The Resource Description Framework (RDF) is a W3C standard [13] for representing information about Web resources. It models the information in the form of triples

Dataset	$\|Triples\|$	$\|Predicates\|$	M K^2-tree	IK^2-tree	RDF-3X
Jamendo	1,049,639	28	0.74	0.74	37.73
Dblp	46,597,620	27	82.48	84.04	1,643.31
Geonames	112,235,492	26	152.20	156.01	3,584.80
DBpedia	232,542,405	39,672	931.44	788.19	9,757.58

Table 1: Space comparison for different RDF datasets (in MB)

(subject, predicate, object), in which the subject represents a resource, the predicate is a property of the subject and the object represents the value of that property for the given subject. Some approaches to represent RDF triples are based in relational databases [17] but multi-indexing native solutions are more frequently used [15]. Vertical partitioning is a usual strategy in this domain. Since the predicate variable has a moderate size, it is used as the partitioning variable. Following this idea, a multiple K^2-tree approach for RDF was already studied and shown to be competitive with the state of the art [2]. In this section we experimentally evaluate the behavior of the IK^2-tree in RDF against the multiple K^2-tree approach and RDF-3X [15], a highly efficient structure for RDF that was shown to be the most competitive alternative [2].

Simple triple patterns explained in Section 3 are the basis of SPARQL, the standard query language for RDF. Therefore, we evaluate how fast simple patterns are solved. We follow a hybrid approach using $K = 4$ in the first five levels of the tree and $K = 2$ in the rest of the levels, for the IK^2-tree and for the multiple K^2-tree. The last levels of the trees (submatrices of size 8×8) are statistically compressed using DACs [6]. Notice that the multiple K^2-tree approach compresses each matrix using an independent vocabulary, which can model the distribution of each adjacency matrix better than the single vocabulary of submatrices 8×8 used in the IK^2-tree. However, the single vocabulary of the IK^2-tree is an advantage if the number of predicates is large because it avoids storing too many vocabularies.

4.1 Experimental framework and results

We include datasets from different domains: Jamendo (`dbtune.org/jamendo`) is a repository of music; Dblp (`dblp.l3s.de/dblp++.php`) stores Computer Science journals and proceedings; Geonames (`download.geonames.org/all-geonames-rdf.zip`) is a geographic database; and DBpedia (`wiki.dbpedia.org/Downloads351`) is an encyclopedic dataset extracted from Wikipedia. Our machine is an AMD-PhenomTM-II X4 955@3.2 GHz, quad-core, 8GBDDR2, running Ubuntu 9.10. The code was developed in C, and compiled using gcc (version 4.4.1) with optimization -O9.

Table 1 shows the size of the different RDF datasets and the compression results obtained by the IK^2-tree, the multiple K^2-tree (MK^2-tree) and RDF-3X. First columns show the number of triples and the number of predicates that each dataset contains. It determines the maximum number of bits of the nodes in the IK^2-tree and the number of individual K^2-trees that have to be created for the multiple K^2-tree.

We can observe that the multiple K^2-tree approach is the most compressed structure for Jamendo, DBLP and Geonames; while for the DBpedia dataset the IK^2-tree achieves the best compression. Since the IK^2-tree contains just a reordering of the

		Geonames			DBpedia		
Category	Pattern	MK^2-tree	IK^2-tree	RDF-3X	MK^2-tree	IK^2-tree	RDF-3X
Bounded Predicate	(S,P,O)	1.8	3.9	2,346.5	3.2	6.2	2,532.4
	(S,P,*)	64.9	110.4	4,882.3	358.7	608.5	4,117.3
	(*,P,O)	0.1	0.3	0.6	0.6	1.6	143.9
	(*,P,*)	0.4	0.5	0.7	0.7	1.6	0.9
Unbounded Predicate	(S,*,O)	5.3	4.4	6,118.6	7,186.1	155.2	6,330.6
	(S,*,*)	95.0	69.7	229.7	3,925.2	911.2	272.3
	(*,*,O)	240.0	187.0	2,473.1	10,918.1	1,444.6	1,377.9

Table 2: Time evaluation of simple patterns for RDF, in μs per result

bits of the individual K^2-tree, their space differences occur because the multiple K^2-tree compresses the last levels of each tree independently, and IK^2-tree uses a global vocabulary, as explained. In DBpedia, the unique vocabulary saves some redundancy, but in the smaller datasets the specific vocabularies obtain better compression. Both representations based on K^2-trees are much more compressed than RDF-3X.

Regarding query efficiency, we test all the basic triple patterns: Table 3 shows the results for Geonames (as a representative domain with few predicates) and for DBpedia (as an example with many predicates). For each simple pattern, we show the average time per query (500 queries were executed for each pattern).

Results show that, for bounded predicates, the multiple K^2-tree is the fastest, and the IK^2-tree is about twice as slow. This is expected, because the navigation in the IK^2-tree is slightly more costly: it must execute additional *rank* operations to compute the number of active bits in each child. In general, RDF-3X is slower.

For patterns with unbounded predicates, instead, the IK^2-tree clearly outperforms the multiple K^2-tree, especially for datasets with many predicates like DBpedia, because it partially solves the problem of the vertical partitioning. RDF-3X only obtains better results for some patterns in DBpedia, but in general it is far less efficient.

4.2 Lazy evaluation

A problem of the navigation algorithms over the IK^2-tree with unbounded predicate is that we have to maintain at each step the list of active predicates. To do this we must check all the bits of each node, and for each *one*, compute its corresponding value using the list of active values of the previous level. For nodes containing many predicates, this mapping can be very expensive. Worse than that, it may not produce any result, because this branch may be discarded in lower levels of the tree. To solve this problem, we propose a *lazy evaluation* strategy that delays the computation of active values of the predicate in each node until a result is found in the branch. This navigation strategy is designed to optimize the performance of queries with unbounded predicate in datasets with many predicates.

In the lazy approach we perform the top-down traversal over the tree without knowing which are the currently active values of predicates (only how many). A later bottom-up mapping will be performed only for the leaves that contain some result. When we reach a leaf we know that we have a result for some values of the predicate. To obtain the actual values for the predicates we start by obtaining the list of active

Simple pattern	Multiple K^2-tree	Interleaved K^2-tree	Lazy IK^2-tree	RDF-3X
(S,*,*)	3,925.2	911.2	232.7	272.3
(*,*,O)	10,918.1	1,444.6	430.8	1,377.9

Table 3: Time, in μs per result, of basic and lazy evaluation in patterns with unbounded predicate on DBpedia

values A in the leaf. Then we recursively update the list of active values, mapping each value with its position in the parent, until we reach the root. At this moment, the elements in A will be the valid predicates for the results of that leaf node.

In addition, we note that for queries where dimensions X or Z are unbounded, we can take advantage of the fact that multiple leaves involved in a query have common ancestors at some level of the tree. When an internal node needs to map several lists of active values from different children we *merge* them so as to check that intermediate node only once. Thus, we start from many lists of active values in the leaves, which are merged when common ancestors are checked in order to avoid redundant mappings.

Table 3 shows how this approach improves the results obtained in RDF for patterns with unbounded predicate in DBpedia. Lazy evaluation improves significantly the results for those queries, achieving better results than RDF-3X in all cases.

5 Representing Temporal Graphs with IK^2-trees

Temporal graphs model the evolution of a binary relation over time, that is, a ternary relation over $X \times Y \times T$, where T (time) is our partitioning variable. A first approach to this problem by using multiple K^2-trees was proposed, where complete snapshots of the graph (or with some kind of differential encoding) were stored [12]. We propose instead to represent each time instant (except t_0) as a change log. At t_0 we store a complete snapshot of the graph (all triples (x, y, t_0)). At t_k, for $k > 0$, we store (x_i, y_j, t_k) iff the relation between x_i and y_j changed between t_{k-1} and t_k, that is, we store for each edge of the graph the time instants when the edge appears or disappears. Using this representation, a relation exists between x_i and y_j at time t_k iff there is an odd number of triples (x_i, y_j, t_m), where $m \in [0..k]$.

Typical queries ask whether elements x and y are related at a given time t_k or during a given interval (t_ℓ, t_r). We use two semantics for intervals: *strong* interval queries ask if the relation existed during the complete interval (in our representation, that means that the relation existed at t_ℓ and no changes occurred in the interval); *weak* interval queries ask if the relation existed at any point within the interval (in our representation, we need to check if any change occurred within the interval or if the conditions for strong queries are fulfilled). All the operations can be easily solved taking into account that each *one* found (except the first bit of each node) is a change, and a relation exists at any time if the number of changes until that time is odd. Using the IK^2-tree, the changes for each (x_i, y_j) are placed in consecutive positions along the bitmap, so fast rank operations can be used in order to count the number of *ones* before a given time and within a given interval. Leaves compressed with DAC are not convenient in this domain since we perform rank operation in all levels.

Dataset	$\|Snapshots\|$	$\|Average\ relations\|$	Multiple K^2-tree	Interleaved K^2-tree
Monkey contact	220	2.500.000	281,11	281,03
Power Law	1.000	2.900.000	4,55	4,54
CommNet	10.000	20.000	137,28	136,51

Table 4: Space comparison on different temporal graphs (in MB)

	CommNet		Monkey Contact		Power Law	
Simple patterns	MK^2-tree	IK^2-tree	MK^2-tree	IK^2-tree	MK^2-tree	IK^2-tree
Dir. Instant	86,78	1,7	0,27	0,14	17,60	1,28
Dir. Weak	87,19	1,84	0,27	0,14	20,72	1,56
Dir. Strong	88,02	1,82	0,26	0,16	19,73	1,44
Rev. Instant	89,07	1,79	0,28	0,15	19,86	1,43
Rev. Weak	90,96	2,07	0,26	0,15	18,36	1,46
Rev. Strong	93,75	2,05	0,26	0,16	19,98	1,50

Table 5: Time comparison on temporal graphs (in msec per query)

5.1 Experimental framework and results

We analyze the temporal evolution of the Monkey Contact dataset (a real social network), CommNet (a random graph where relations exist during a short period of time) and a power-law degree distributed graph. We represent these datasets using our IK^2-tree, and also using multiple K^2-trees that encode the same change logs. Table 4 shows the number of snapshots and the average number of changes (edges) per snapshot, and the size of the representations obtained with the IK^2-tree and the multiple K^2-tree. These are almost identical (small differences are due to redundant information stored in each of the K^2-trees).

Table 5 shows the query performance achieved for the three datasets with the IK^2-tree, in contrast with the multiple K^2-tree approach. We ask for direct and reverse neighbors of a given node at a time instant or time interval with strong and weak semantics. We run sets of 2000 queries of each type, where nodes and time instants are chosen randomly, while the length of the intervals is fixed (100). The experimental results show the superiority of the Interleaved K^2-tree regarding the multiple K^2-tree approach in terms of querying efficiency in the context of temporal graphs.

6 Conclusions

IK^2-tree is a promising data structure for ternary relation representation providing, in very compact space storage, indexing capabilities over the three dimensions. Our experimental results on real-world applications show its superiority over the state-of-the-art vertical partitioning schemes.

The full potential of IK^2-tree is still unexplored. For example, fact that each node of the tree stores a bitmap instead of a single bit is interesting in cases, such as temporal graphs, where counting the number of bits (1 or 0) between two positions (changes between two instants) provides useful information. We plan to pursue on further functionality and other applications of this data structure.

References

[1] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *Proc. 33rd VLDB*, pages 411–422, 2007.

[2] S. Álvarez-García, N. R. Brisaboa, J. D. Fernández, and M. A. Martínez Prieto. Compressed k2-triples for full-in-memory RDF engines. In *Proc. 17th AMCIS*, 2011.

[3] A. Apostolico and G. Drovandi. Graph compression by BFS. *Algorithms*, 2(3):1031–1044, 2009.

[4] J. Barbay, L. C. Aleardi, M. He, and J. I. Munro. Succinct representation of labeled graphs. *Algorithmica*, 62(1-2):224–257, 2012.

[5] Paolo Boldi and Sebastiano Vigna. The webgraph framework i: compression techniques. In *Proceedings of the 13th international conference on World Wide Web*, pages 595–602. ACM, 2004.

[6] N. Brisaboa, S. Ladra, and G. Navarro. DACs: Bringing direct access to variable-length codes. *Inf. Proc. Manag.*, 49(1), 2013.

[7] N. R. Brisaboa, G. de Bernardo, and G. Navarro. Compressed dynamic binary relations. In *Proc. 22nd DCC*, pages 52–61, 2012.

[8] N. R. Brisaboa, S. Ladra, and G. Navarro. K2-trees for compact Web graph representation. In *Proc. 16th SPIRE*, pages 18–30, 2009.

[9] D. Chakrabarti and C. Faloutsos. Graph mining: Laws, generators, and algorithms. *ACM Computing Surveys (CSUR)*, 38(1):2, 2006.

[10] F. Claude and G. Navarro. Fast and compact web graph representations. *ACM Trans. Web*, 4(4):16, 2010.

[11] G. de Bernardo, S. Álvarez-García, N. Brisaboa, G. Navarro, and O. Pedreira. Compact querieable representations of raster data. In *Proc. 20th SPIRE*, pages 96–108, 2013.

[12] G. de Bernardo, N. R. Brisaboa, D. Caro, and M. Andrea Rodríguez. Compact data structures for temporal graphs. In *Proc. 23rd DCC*, 2013.

[13] F. Manola and E. Miller, editors. *RDF Primer*. W3C Recommendation. 2004. `http://www.w3.org/TR/rdf-primer/`.

[14] J. I. Munro. Tables. In *Proc. 16th FSTTCS*, pages 37–42, 1996.

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

[16] N. Ritter and M. Ruth. The GeoTiff data interchange standard for raster geographic images. *Int. J. Remote Sensing*, 18(7):1637–1647, 1997.

[17] S. Sakr and G. Al-Naymat. Relational processing of RDF queries: a survey. *SIGMOD Rec.*, 38(4):23–28, 2010.

[18] X. Yan, P. S. Yu, and J. Han. Graph indexing: a frequent structure-based approach. In *Proc. 2004 ACM SIGMOD*, pages 335–346. ACM, 2004.