Compressed Dynamic Binary Relations[*]

Nieves R.Brisaboa[+], Guillermo de Bernardo[+], and Gonzalo Navarro[†]

[+]Database Laboratory
University of A Coruña
{brisaboa, gdebernardo}@udc.es

[†]Dept. of Computer Science
University of Chile
gnavarro@dcc.uchile.cl

Abstract We introduce a dynamic data structure for the compact representation of binary relations $\mathcal{R} \subseteq A \times B$. Apart from checking whether two objects $(a, b) \in A \times B$ are related, and listing the objects of B related to some $a \in A$ and vice versa, the structure allows inserting and deleting pairs (a, b) in the relation, as well as modifying the base sets A and B. The data structure is a dynamic variant of the k^2-tree, a static compact representation that takes advantage of clustering in the binary relation to achieve compression. We apply our dynamic data structure to the representation of Web graphs and RDF databases, showing that it combines good compression ratios with fast query and update times.

1 Introduction

Binary relations arise everywhere in Computer Science: graphs, matchings, discrete grids, inverted indexes, and pairs of database columns are just some examples. Formally, a binary relation between two sets A and B is a subset $\mathcal{R} \subseteq A \times B$. Typical operations of interest are: determine whether a pair (a, b) is in $\mathcal{R}$, find all the elements $b \in B$ such that $(a, b) \in \mathcal{R}$, given $a \in A$, and vice versa. More sophisticated ones aim, for example, at retrieving all pairs $(a, b) \in \mathcal{R}$ where $a \in [a_1, a_2]$ and $b \in [b_1, b_2]$. For example, on a Web graph (where nodes are Web pages and relations are hyperlinks) the above operations determine whether a link exists between two pages, find the direct or reverse neighbors of a node, and find all the links between two Web sites. On an inverted index, these allow one to determine whether a word appears in a document, to list the documents where a word appears, to list the local vocabulary of a document, and the more sophisticated one enables on-the-fly stemming as well as hierarchical or versioned document collections.

Two two natural ways to represent binary relations extend graph representations: a binary adjacency matrix or an adjacency list. On large binary relations, reducing space while retaining functionality is crucial in order to operate efficiently in main

[*]NB and GdB were partially funded by MICINN (PGE and FEDER) grants TIN2009-14560-C03-02, TIN2010-21246-C02-01 and CDTI CEN-20091048, and by Xunta de Galicia (co-funded with FEDER) ref. 2010/17. GdB was also funded by MICINN ref. BES-2010-033262 (FPI program). GN was partially funded by the Millennium Institute for Cell Dynamics and Biotechnology (ICDB), Grant ICM P05-001-F, Mideplan, Chile.

memory. There has been work on compressing general binary relations [2], as well as specific ones such as Web graphs [3].

Brisaboa et al. [5] introduced a compact data structure called *k^2-tree*. It was initially proposed for the compression of Web graphs, where it was shown to be very competitive (see also [6]). Since then, it has also been successfully applied to other domains such as RDF databases [1] and social networks [6]. In fact, k^2-trees can be used for the representation of general binary relations and take advantage of clustering in the binary matrix to achieve compression. They support elegantly all the described operations (simple and sophisticated) as instances of a more general query.

However, just like the other compressed representations of graphs and binary relations, k^2-trees are essentially static. This discourages their use in cases where the binary relation changes due to the insertion or deletion of pairs (a, b) (e.g., adding or removing edges in a graph) or of elements in A and B (e.g., adding or removing graph nodes, or words or documents in inverted indexes).

In this paper we introduce the *dk^2-tree*, a dynamic version of the k^2-tree. This structure achieves space utilization close to that of the static structure, and allows the insertion and deletion of pairs and elements in the sets (i.e., changing bits and inserting/deleting rows/columns in the binary matrix). Our experiments show that dk^2-trees achieve good space/time tradeoffs in comparison with the equivalent static representation. We also show that the added capabilities of the dynamic structure make it suitable for the representation of dynamic binary matrices.

2 k^2-trees

A k^2-tree is conceptually a k^2-ary tree that corresponds to a recursive partition of a binary matrix. At each partitioning step, the current matrix of size $n \times n$ is divided in k^2 submatrices of size $n/k \times n/k$. Figure 1 shows an example of the division of a binary matrix using a k^2-tree, for k=2. The submatrices are numbered from 0 to k^2-1, starting from left to right and top to bottom. The first level of the tree contains one node with k^2 children, representing the k^2 submatrices in which the original matrix is divided. Each of these children is represented using a single bit: 1 if the submatrix has at least one cell with value 1, or 0 otherwise. A 0 child means that there are no ones in the corresponding submatrix, so its children are not represented in the next level. The method proceeds recursively for each 1 child until the current submatrix is full of zeros or we reach the basic cells of the original matrix.

To access a cell of the matrix, the tree is navigated from the root until a 0 is found or the last level is reached. Starting at the root, one of the k^2 children is selected at each level, depending on the submatrix that we want to access. If the value for that node is 0 we have found an empty submatrix, and navigation ends. If the value is 1, we proceed recursively to the next level accessing one of its k^2 children. For example, if we want to access the highlighted position in Figure 1 (row 9, column 7), we start at the root of the k2-tree. From the row and column we are looking for we know that the desired cell is in the third submatrix, so we proceed to the third child. As it contains a 1, we continue the search at the next level. We repeat the process until we reach a zero node or the desired cell of the k^2-tree.

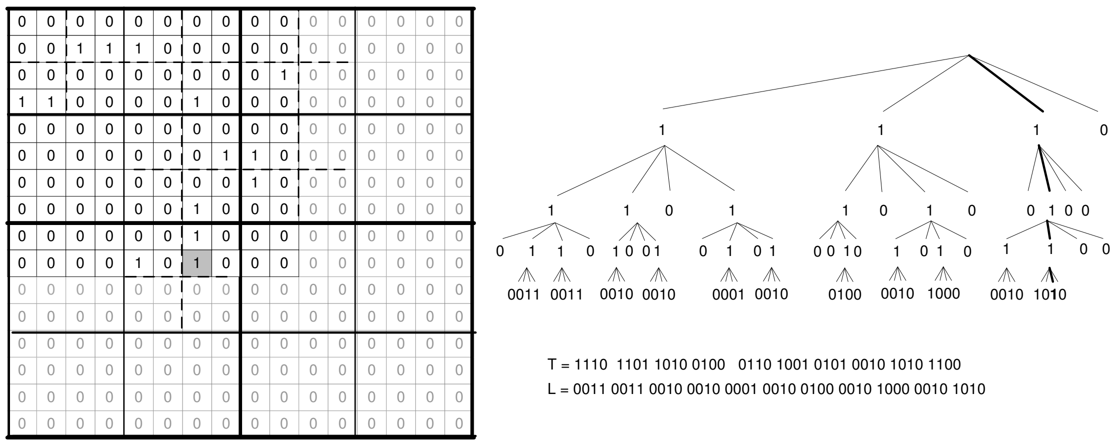

T = 1110 1101 1010 0100 0110 1001 0101 0010 1010 1100
L = 0011 0011 0010 0010 0001 0010 0100 0010 1000 0010 1010

Figure 1: example of k^2-tree for a binary matrix

This conceptual tree is implemented using two bit arrays: **T** (Tree) contains the bits for all the levels of the tree except for the last one. The bits are taken in a levelwise traversal of the conceptual tree. **L** (Leaves) stores the bits of the last level of the tree.

Starting at a given node (a position *pos* in the bit array T), it is easy to see that the k^2 children of that node will be at $pos' = rank_1(T, pos) \times k^2$, because each bit set to one in a level add k^2 bits to the next level and bits set to zero do not have descendants. A rank structure is built over T to provide an efficient $rank_1$ operation. Following the same example in Figure 1, if we want to navigate to the third child of the root we access position 2 in T (we start numbering positions in 0). It contains a 1, so we will move to position $rank_1(T, 2) \times 4 = 3 \times 4 = 12$ in the bitmap T. That node contains its k^2 children, with values 0-1-0-0, in positions 12-15. The cell we are looking for is in the second submatrix, so we access the second child at position 13, and again navigate to $rank_1(T, 13) \times 4 = 32$. We continue this procedure until we find a 0 or we reach the leaf.

In addition to the retrieval of a single cell of the matrix, k^2-trees can perform other operations efficiently. To find all the ones in a row/column, we can modify the basic search so that at each level of the k^2-tree, instead of accessing a single child, we access the k submatrices of the node that overlap that row(column). With some additional calculations, the k^2-tree can also retrieve any range [u1,u2]-[v1,v2] efficiently.

Several enhancements have been proposed over this first approach. The first modification is the use of different k values at each level of the k^2-tree. Using a bigger k for the first levels and smaller for the remaining ones, one can achieve better query times (because the k^2-tree's height is reduced) while keeping the space close to the optimum. In [7] a compression method for the bitmap L is proposed. In this approach, the lowest levels of the k^2-tree are grouped, yielding submatrices of size bigger than k (ie. 8×8 instead of 2×2). These small submatrices are compressed using variable length codes according to their frequency in the represented matrix. The bitmap L is replaced with a matrix vocabulary that contains the submatrices sorted by frequency and the encoded sequence of matrix identifiers. The sequence is encoded using Direct Access Codes[4] to provide direct access to any position of the sequence. This variant drastically increased the compression of Web Graphs, showing similar query times.

3 Dynamic k^2-tree implementation: dk^2-trees

The conceptual k^2-tree is represented in the static version using the T array for the internal nodes and the L array for the leaves of the k^2-tree. In our dynamic implementation, we represent T and L with two trees, that we call TTree and LTree. Our dynamic approach to represent the k^2-tree using these trees is called dk^2-tree. Notice that the dk^2-tree is not a single tree but a way to implement the conceptual k^2-tree using TTree and LTree, that are the real tree structures we use. The leaves of TTree and LTree contain rougly the same representation of the bitmaps T and L of a static k^2-tree implementation. The internal nodes will provide access to arbitrary positions and will also act as a dynamic rank structure. Our representation is in fact a practical implementation of a dynamic bit vector[8] over T and L.

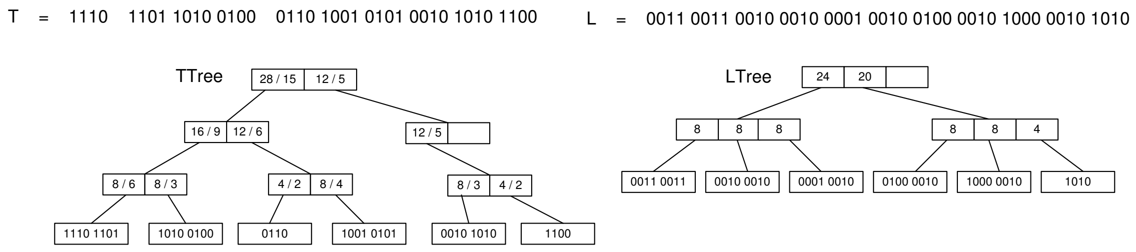


Figure 2: Dynamic k^2-tree representation

Figure 2 shows the dk^2-tree representation for the matrix of Figure 1. Given the bitmaps T and L of a static k^2-tree, we build over them the trees TTree and LTree. First, we layout the bitmaps T and L in blocks of up to B bits, being B the parameter for the node size. The nodes of our trees may not be completely full, because we will split nodes when updating the tree. The internal nodes of our trees contain a set of entries that are used to access the leaves. Each entry contains, in addition to the pointer to the child, some information to provide the basic navigation operations. In TTree, each entry in the internal nodes contains the total number of bits (bit counters) and number of ones (ones counters) in its descendant leaves. As we move up TTree, the bit and rank counters are calculated adding up the counters of the pointed child. Internal nodes for LTree cointain only the bit counters and the pointer to the child, because as in static k^2-trees we do not need to perform rank operations in L.

3.1 Navigation

To navigate the k^2-tree using TTree and LTree, we proceed as we would in a static k^2-tree representation. At each level of the conceptual k^2-tree, we must access a child of the current node in the conceptual k^2-tree, that is, access a position in the bitmaps T or L. In dk^2-trees, to access a position in T or L we must find the leaf of TTree or LTree that contains that position. This is performed by the operation *findLeaf*, shown in Procedure 1, that uses the bit and ones counters in the internal nodes to find the leaf for the searched position. When we reach the leaf, we access the bit we are looking for (we know the offset in the leaf because *findLeaf* has already calculated the number of bits before the beginning of the leaf). If the bit we access has value 0, we end our navigation. If it is 1, we must calculate the rank value up to that position. Again, we already know the rank value at the beginning of the leaf using the ones counters, so we just need to perform the rank operation inside the leaf (*rankLeaf* operation). Procedure 2 shows the complete process to access a single cell of the matrix. All the operations supported by static k^2-trees can be performed in the dk^2-tree implementation in the same way.

Proc. 1 findLeaf(node, pos)

```
if not isLeaf($node$) then
   $i = 0$
   $data$ = readData($node$)
   while ($accBits + data[i].bits \leq pos$)
   do
      $accOnes$ += $data[i].ones$
      $accBits$ += $data[i].bits$
      $child = data[i].child$
      $i = i + 1$
   end while
   return findLeaf($child$, $pos$)
else
   return ($node$, $accBits$, $accOnes$)
end if
```

Proc. 2 findCell(row, col)

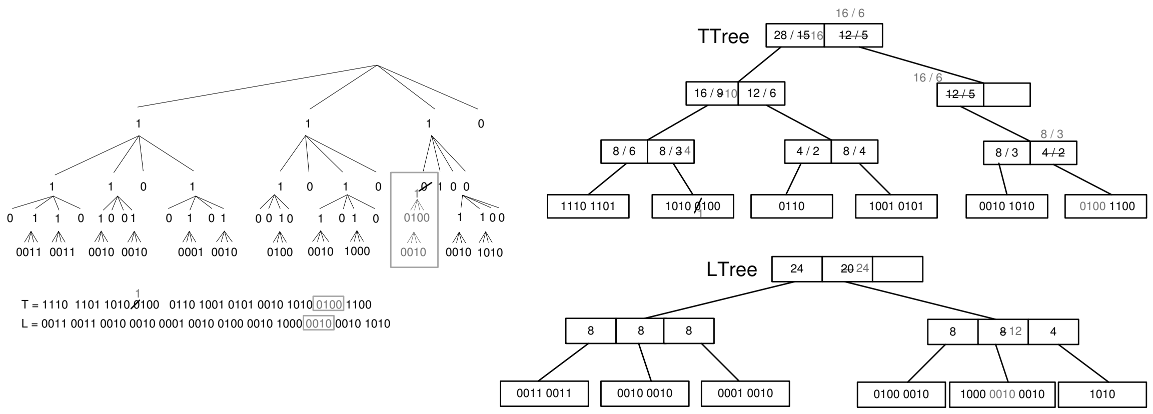
```
1: pos = 0
2: for $level = 1 \to l - 1$ do
3:    pos += CalculateChild(row,col,level)
4:    ($data$, $accBits$, $accOnes$) = findLeaf(TTree.root, pos)
5:    if bitget($data$, $pos - accBits$) then
6:       rank = $acumOnes$ + rankLeaf($data$, $pos - accBits$);
7:       pos = $rank \times k^2$
8:    else
9:       return 0
10:   end if
11: end for
12: pos += CalculateChild(row, col, l)
13: pos -= GetTotalSize(T)
14: ($data$, $accBits$) = findLeaf(LTree.root, pos)
15: return bitget($data$, $pos - accBits$)
```

The cost of operations in dk^2-trees can be calculated in relation to the cost in a static implementation. The time cost for a single link retrieval in our structure can be separated into $(l - 1) \times rankLeaf + l \times findLeaf$, where l is the number of levels of the conceptual k^2-tree. To speed up the $rankLeaf$ operation, we implemented a simple rank structure, similar to the one used in the static implementation of k^2-trees, that is added to the leaves of TTree. Thus, the main overhead in dk^2-trees is the $findLeaf$ operation. This operation requires traversing a full path in TTree or LTree for each level of the conceptual k^2-tree. However, in practice all operations start at the leftmost leaf of TTree (the root of the conceptual k^2-tree) and continue accessing different leaves of TTree and LTree from left to right. The $findLeaf$ operation can start the search from the previous leaf accessed, making this search very efficient at least for the first levels. Following this approach the cost of each $findLeaf$ operation depends on the distance between the current leaf and the next we need to access, because if they are close the will share their parent node or a close ancestor, and we will save the traversal from the root of the tree to that common ancestor.

3.2 Updating dk^2-trees

dk^2-trees support the creation and deletion of relations (change zeros into ones and vice versa) and elements (adding or removing rows/columns to the matrix).

When replacing a zero with a one in a conceptual k^2-tree, we start by descending from the root of the k^2-tree choosing always the branch that leads us to the proper cell. If we reach the last level of the k^2-tree (represented in our LTree), we just need to change the value of the position from 0 to 1 (change a bit in a leaf of LTree). This situation will only happen if the matrix already contained a cell with a 1 in a position very close to the position we want to set to 1. Figure 3(left) shows a more general case: at some level we find a zero in the branch that would lead us to the cell we are changing to 1. From this point, we need to create a new path in the k^2-tree. The new branch in the k^2-tree is represented within a rectangle in Figure 3. This operation can be translated in the dk^2-tree (right side of the figure) as a single operation of setting a bit to 1 in a leaf of TTree, zero or more operations of adding k^2 bits to leaves of TTree and one operation of adding k^2 bits to a leaf of LTree. Additionally, the bits and ones counters in the internal nodes must be updated whenever we change a leaf of TTree or LTree.



T = 1110 1101 1010 ~~0~~100 0110 1001 0101 0010 1010 0100 1100
L = 0011 0011 0010 0010 0001 0010 0100 0010 1000 0010 0010 1010
TTree
LTree

Figure 3: Adding a new one to position (9, 2) of the matrix

When a leaf of TTree or LTree reaches the maximum node size we split it in two new nodes, always keeping groups of k^2 siblings in the same leaf. This is propagated to the internal nodes, causing the insertion of a new entry in the parent, and eventually new overflows in the internal nodes. To achieve better space utilization we allow partial expansions of a node before splitting it. For instance, with a a base node of 512 bytes and three partial expansions we can expand it to 640, 768 and 896 bytes before splitting. When a fully expanded node overflows, it is split in two nodes of 512 bytes.

The deletion behaves nearly like the insertion. In this case, we first perform a search of the complete path in the conceptual k^2-tree, until we reach the position that we want to set to 0, that will be in a leaf of LTree. We set the value of that cell to 0, and check its k^2-1 siblings. If all of them are 0, the current branch of the k2-tree must be deleted, and we repeat the process up to the upper level of the k^2-tree, affecting now TTree.

The insertion of new elements (rows and/or columns) is done using the free rows and columns we have. The number of rows/columns in a binary matrix is rounded up to a power of k to represent it with a k^2-tree, so many rows and columns are already created but left empty and ready to be used. When all the rows(columns) in a $n \times n$ matrix have been used, this matrix is expanded to size $kn \times kn$. This causes the addition of a new root level in the k^2-tree with k^2 children, where the first of them is the old root. Therefore, to add this new root level to the k^2-tree we just need to add k^2 bits at the beginning of the leftmost leaf of TTree, handling overflows and updating the path to the root as we would when we insert a new one.

The deletion of rows/columns is also implemented trivially. When a row/column is deleted, all the ones belonging to it are deleted, and its identifier is added to a free-list. Deleted rows may be reused if new rows are inserted afterwards.

3.3 Compression of L

The compression of the bitmap L with a matrix vocabulary obtained very good results in static k^2-trees. In dk^2-trees, the leaves in LTree could be encoded using the same encoding procedure. However, when we update the k^2-tree we need to check whether a given matrix is already in the vocabulary or not, that is, we need this dictionary to be searchable. This need produces a space overhead the static k^2-trees do not have.

We implemented a variant of dk^2-tree with a dictionary that contains in addition to the matrix vocabulary a hash table to search this vocabulary. We also store the frequency for each submatrix in a frequency array. This helps keeping the vocabulary size small and allows us to know how good the compression is. Our dictionary uses a small set of submatrices to build an initial vocabulary and then adds new matrices as needed. When a matrix has frequency 0 it is deleted from the vocabulary and its code can be reused. Nevertheless, it can be seen that the overhead added to the size of the vocabulary is quite important.

To ensure that the compression ratio is good, additional tests can be performed using the frequency array. If the compression obtained by our current dictionary worsens too much we can simply compute the optimal values for the vocabulary using the frequency array, and then rebuild LTree in a single traversal, replacing the old codes that encode L with the optimum ones. To determine when LTree should be rebuilt, we can use simple heuristics (we can rebuild always every p insertions, or count the number of matrices that are out of the order they would have if ordered by their actual frequency). If we wish to guarantee that the compression of L is never too far from the optimum, we can even keep track of the actual optimum vocabulary, getting an important overhead in the total size of our vocabulary but ensuring that the compression of LTree is close to the optimum.

4 Experimental evaluation

The space utilization of dk^2-trees, without any compression of the bitmap L, is never too bad in comparison with the static k^2-trees. Both structures store the same basic information, the bitmaps T and L, but while static k^2-trees use static rank structures to navigate these bitmaps, in dk^2-trees we need the internal nodes of TTree and LTree to store not only the ones counters used in the rank but also the bit counters. There are two main causes that make dk^2-trees bigger than static k^2-trees: first, the internal fragmentation of the nodes in TTree and LTree, and second, the overhead needed in dk^2-trees to maintain the matrix vocabulary searchable.

We tested dk^2-trees in comparison with a static implementation in two different contexts: the representation of Web Graphs and of RDF graphs.

We run all our experiments in a machine with 4 Intel(R) Xeon(R) E5520 CPU cores at 2.27 GHz 8 MB cache and 72 GB of RAM memory. The machine runs

Ubuntu GNU/Linux version 9.10 with kernel 2.6.31-19-server (64 bits). Our code is compiled using gcc 4.4.1, with the -O9 directive.

4.1 Web Graph representation

We compare our dk^2-tree approach with the static one for the representation of Web Graphs. We test both implementations using a matrix vocabulary for the compression of L (*dyn-comp* for dk^2-trees, *static-comp* for static k^2-tres) and without compressing L (*dyn-plain* and *static-plain*). Our TTrees and LTrees use a base node size of 512 bytes, with 3 partial expansions to get a good utilization of the nodes of the trees. Additionally, dk^2-trees are built by repeated insertion, so the space utilization of the nodes is not optimized. We also use a rank structure in the leaves of TTree to speed up the rank operations. To compress L, we use the optimum values for the submatrix size for each case (8×8 matrices for static k^2-trees, and 4×4 matrices for dk^2-trees).

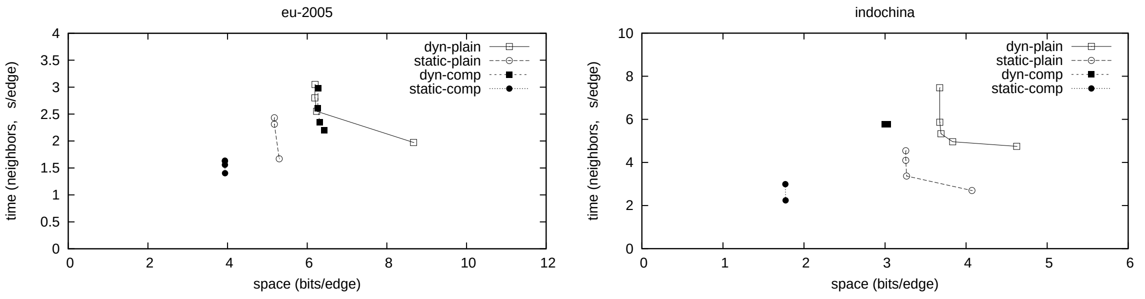


Figure 4: Space time comparison for the collections eu-2005 and indochina

Figure 4 shows a comparison in space and query time for two web graph datasets: *eu-2005*, that contains 19 million edges, and *indochina*, with 194 million. We show the space in bits per edge of the represented web graph, and the time of the main operation in the context of Web Graphs: neighbor retrieval (retrieving all the ones in a row of the matrix), measured in μs/edge. For each approach, the space/time evolution is given by different values of k (we use the hybrid approach with k=4 for some of the first levels and 2 for the remaining, getting a space/time tradeoff depending on how many levels have the bigger k value). With the *plain* representation, dk^2-trees are close in space to the equivalent static k^2-trees, but static k^2-trees achieve much better compression ratios when they compress L, being roughly 2 times smaller than our approach. In dk^2-trees, if we use a matrix vocabulary we do not get the same improvement: for the collection indochina (and in general for bigger web graphs), we gain in compression, but we are still far from the improvement of static k^2-trees; for the collection eu-2005, the version compressing L achieves the same size that the plain version.

4.2 Representation of RDF databases

In [1] k^2-trees are used to represent RDF datasets grouping the RDF triples by predicate and using a k^2-tree to represent the relations for each predicate. This partition by predicate, called *vertical partitioning*, is a common approach that permits representing the database as binary relations between subjects and objects for each predicate.

We compare dk^2-trees in terms of space and time with their static implementation, following the same approach of building a dk^2-tree to represent each predicate.

Table 1 shows the results of space for different RDF collections. We show the number of triples and predicates in each collection, and the plain size of the pairs (subject,object) per predicate, where each subject and object is represented by a 32 bit integer. The last two columns show the compression obtained by static k^2-trees and the equivalent dk^2-tree. Static k^2-trees use a matrix vocabulary, with 8×8 leaves. For dk^2-trees, we show the result with no compression of L. For the collection dbtune we also show in parentheses the results compressing L with a vocabulary of 8×8 matrices, that improves the compression by 10% (in the other collections using a matrix vocabulary does not improve the results of the plain version). It can be observed that the dk^2-tree representation achieves compression close to the static structures in all the datasets.

Collection	#triples	#predicates	Plain size	k^2-tree size	dk^2-tree size
geonames	9,415,253	20	71.83	17.81	25.14
dbtune	58,920,361	394	449.53	152.38	212.52 (188.13)
dbpedia	232,542,405	39672	1774.46	878.04	1060.59

Table 1: Compression results for RDF collections (sizes in MB)

After this, we implement the basic RDF queries in our structure, those that involve a single query pattern of the form (s,p,o). They are divided according to the elements that are fixed and free in the triple requested. We denote with ?S, ?P and ?O a free subject, predicate or object, and with S, P, O a fixed one. The different patterns correspond to different k^2-tree operations: (S,P,O) requires a single cell retrieval operation; (S,P,?O) and (?S,P,O) accessing a row or column; (?S,P,?O) involves retrieving all the ones in a matrix. The same operations with a free predicate involve the same operation in all the k^2-trees of the collection.

We run a set of queries of each type over the collection dbpedia. These queries are selected so that predicates with many and few relations are accessed (typically in RDF graphs the distribution of relations among the predicates is skewed, because some predicates are much more frequent than others). We also choose queries in which the number of results varies from a single result to several hundreds. The average number of relations per object or subject is not very high, so usually only the (?S,P,?O) queries will return a higher number of results. Table 2 shows the results for the query time in ms/query. The efficiency of dk^2-trees to answer basic queries is again relatively close to that of a static k^2-tree. The single cell retrieval operations are several times slower but in operations with a free subject or a free object the time cost over the dynamic trees gets closer to that of static k^2-trees. In these queries, the time per query in dk^2-trees is never over the double of the time in the static representation.

	(S,P,O)	(S,P,?O)	(?S,P,O)	(?S,P,?O)	(S,?P,O)	(S,?P,?O)	(?S,?P,O)
static	0.00085	0.682	0.432	4.433	7.328	74.394	49.397
dk^2-trees	0.00282	1.214	0.836	8.081	26.372	115.49	55.512

Table 2: Query times in dataset dbpedia(ms/query)

The dk^2-tree representation does not only provide good compression ratios, but also provides the required functionalities to maintain a fully functional RDF database.

Modifications in the triples stored in the database can be performed as insertions and deletions in the appropriate dk^2-trees. New predicates just require the creation of a new empty dk^2-tree. New subjects and objects require the expansion of the represented matrices to add a new row or column to all the dk^2-trees. Therefore, our representation can provide all the basic operations needed in a RDF database, with a compression rate close to that of a static k^2-tree representation.

5 Conclusions

The representation of binary relations can be useful in many contexts. In many cases, there is a need to provide a dynamic representation for these relations. We have presented dk^2-trees, a dynamic version of a inherently static data structure, the k^2-tree, that provides the dynamic operations needed in many contexts: modification of the contents of the matrix and addition or deletion of rows/columns. dk^2-trees can operate in space close to that of static k^2-trees, that are currently state-of-the-art in Web Graph compression and can achieve good results in the compression of RDF graphs. Particularly, dk^2-trees add all the required capabilities for a RDF database to the good compression and query times obtained by k^2-trees.

More experimentation is needed to achieve compression ratios closer to that of the best static k^2-trees, because the matrix vocabulary for the compression of L does not achieve so good results in dk^2-trees. We also intend to test thoroughly the efficiency of our data structure when working in external memory. Our tree is suitable for its direct utilization from external memory, and we expect that the good compression achieved by dk^2-trees will allow the caching of a significant part of our tree, providing a good I/O efficiency in practice.

References

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

[2] J. Barbay, F. Claude and G. Navarro. Compact Rich-Functional Binary Relation Representations. In *LATIN*, pages 170–183. Springer, 2010.

[3] P. Boldi and S. Vigna. The WebGraph Framework I: Compression techniques. In *WWW*, pages 595–601. ACM Press, 2003.

[4] N. R. Brisaboa, S. Ladra, and G. Navarro. Directly addressable variable-length codes. In *SPIRE*, pages 122–130, 2009.

[5] N. R. Brisaboa, S. Ladra, and G. Navarro. k2-trees for compact web graph representation. In *SPIRE*, pages 18–30, 2009.

[6] F. Claude and S. Ladra. Practical representations for web and social graphs. In *CIKM*, pages 1185–1190, 2011.

[7] S. Ladra. *Algorithms and Compressed Data Structures for Information Retrieval.* PhD thesis.

[8] R. Raman, V. Raman, and S. S. Rao. Succinct dynamic data structures. In *WADS*, pages 426–437, 2001.