This is a post-peer-review, pre-copyedit version of an article published in Knowledge and Information Systems. The final authenticated version is available online at: http://dx.doi.org/10.1007/s10115-018-1275-x

Compact and Efficient Representation of General Graph Databases[1]

Sandra Álvarez-García[1], Borja Freire[2], Susana Ladra[3] and Óscar Pedreira[3]

[1]Indra, A Coruña, Spain; [2] Enxenio S.L., A Coruña, Spain; [3] Universidade da Coruña, CITIC, Database Laboratory, A Coruña, Spain

Abstract. In this paper, we propose a compact data structure to store labeled attributed graphs based on the k^2-tree, which is a very compact data structure designed to represent a simple directed graph. The idea we propose can be seen as an extension of the k^2-tree to support property graphs. In addition to the static approach, we also propose a dynamic version of the storage representation, which allows flexible schemas and insertion or deletion of data. We provide an implementation of a basic set of operations, which can be combined to form complex queries over these graphs with attributes. We evaluate the performance of our proposal with existing graph database systems and prove that our compact attributed graph representation obtains also competitive time results.

Keywords: Compression; Graph Databases; Property Graphs; Attributed Graphs; Compact Data Structures; Dynamic Graphs

1. Introduction

Graphs are a natural way for modeling data in such domains where the most relevant information relies on the relationships between the entities. Some representative examples are Web graphs (Raghavan and Garcia-Molina 2003), social networks (Padrol-Sureda et al. 2010), computational biology (Böhm and Schneider 2000), pattern recognition (Conte et al. 2004), chemical data analysis (Aggarwal and Wang 2010), and geographic information systems, among others. In the last years, many research lines have emerged focusing on the analysis of

[1] A preliminary partial version of this paper appeared in Proceedings of the Eighth Workshop on Mining and Learning with Graphs (MLG2010), pp. 18–25, 2010.

Received 24 Feb 2016
Revised 24 May 2018
Accepted 31 Jul 2018

data showing this graph nature. Furthermore, in the Big Data Era, huge volumes of data are generated every day. This information needs to be stored and processed efficiently in terms of space and time. In this scenario, where graph mining involves complex analyses on huge datasets, the design of new compact graph representations that can be accessed efficiently has become an important research field.

In many cases, the graph models used to represent the relevant information for a domain are simple directed graphs. Many compact and efficient proposals have appeared to represent this kind of graph (Jacobson 1989, Boldi and Vigna 2004, Chierichetti et al. 2009, Claude and Navarro 2010, Hernández and Navarro 2014, Fischer and Peters 2016, Maneth and Peternek 2016). However, in many cases, this model is not enough, because nodes and edges contain complex information that must be stored and accessed. These domains where nodes and edges include a set of attributes (key/value) define a new model of graphs, usually called *attributed graphs* or *property graphs*. For this scenario, *graph database models* emerged to give a theoretical support to attributed graphs. These new models are characterized by representing the schema, the data, the queries, and the results as a graph (Larriba-Pey et al. 2014). Built over those theoretical models, many practical Graph Databases Engines have been proposed (Ciglan et al. 2012). DEX (Martínez-Bazan et al. 2012, Martínez-Bazan et al. 2007) or Neo4j (Han et al. 2011) are two relevant examples.

Given the amount of information that those graph database engines have to manage, it is important to focus on the design of efficient and compact structures to represent attributed graphs, as it improves their management and querying. In many domains, a plain representation of the graph may not fit in main memory, and swapping can degrade the performance. Typical access patterns and navigation over the graphs also make difficult an out-of-core processing of graph data. Therefore, it will be important to compress and index the dataset, so it can be stored in main memory and support efficiently the navigation operations, without the need of uncompressing during the analysis of the dataset. In this article, we propose a compact structure to store attributed graphs, whose internal representation is based on the k^2-tree (Brisaboa et al. 2014), a static structure designed to represent simple directed graphs (binary relations) in main memory. Our goal is to study the possibility of extending this compact structure to obtain a very succinct representation of attributed graphs that supports efficient graph operations and access to the attributes of the nodes and edges of the graph. The k^2-tree has been successfully extended in the past, for instance by proposing a dynamic variant that supported changes in the set of edges (Brisaboa et al. 2017), or to support other types of data representation, such as temporal graphs (Caro et al. 2016, Álvarez-García et al. 2017), RDF datasets (Álvarez-García et al. 2017), or raster data (de Bernardo et al. 2013, Ladra et al. 2017). For instance, RDF datasets can be considered ternary relations, thus, they can be decomposed into a collection of binary relations, and represented with a variant of the k^2-tree that provides indexing capabilities in all the three dimensions (Álvarez-García et al. 2017). Raster data can be represented using also a collection of k^2-trees, one per each different value existing at the raster dataset (de Bernardo et al. 2013), or by generalizing the original method for integer values instead of bit values (Ladra et al. 2017). However, none of the previous works tried to extend the k^2-tree structure to represent general graphs, including labels and attributes.

The result of this work is the *AttK^2-tree*, a compact structure to store attributed graphs based on the representation of binary relations in a very compact way using the k^2-tree structure. In addition, we present a dynamic version, which allows changing both the schema and the data contained in the database. We denote it as *dynAttK^2-tree*. We compare our proposals with other attributed graph representations in the state of the art, obtaining the best space/time trade-off for basic query operations.

The paper is structured as follows. Section 2 describes the most representative tools existing in the state of the art to manage attributed graphs. Section 3 briefly reviews the k^2-tree, which is used in our proposal. In Section 4, we present our compact data structure to store such graphs with attributes, that is, the AttK^2-tree. This section focuses on the physical storage. Section 5 presents the operations implemented in the AttK^2-tree. Section 6 presents the dynamic variant of our proposal, that is, dynAttK^2-tree. Finally, Section 7 provides an experimental evaluation of the system using representative cases of study where we compare our proposal with other attributed graph representations in the state of the art.

2. Systems for attributed graphs

Last years, graph database models have been proposed to represent attributed graphs. These models specify the data, the queries, the results and, in many cases, even the schema of the graph as a graph (Angles and Gutiérrez 2008). Many theoretical models and their corresponding query languages were proposed to represent and navigate graphs. Some examples are the *Hypernode Model* (Levene and Poulovassilis 1990), whose main feature is that nodes can be graphs by themselves, and GOOD (Graph Oriented Database Model)(Gyssens et al. 1994), where data manipulation operations (insertions and deletions of nodes and edges, and clustering of nodes depending on some properties) are specified as graph transformations.

Built over those theoretical models, many practical *graph databases engines* have been proposed. In this section, we describe some of the most relevant works in this area, including their internal data structures for graph representation and processing. Some of the graph database systems reviewed in this section use storage data structures specifically designed for graphs, as it is the case of DEX, Neo4j, or HyperGraph, while others, such as SAP HANA Graph and SQLGraph, store graphs in relational database tables, or in a combination of relational databases with complementary stores.

2.1. DEX

DEX (now named as Sparksee[2]) (Martínez-Bazan et al. 2012, Martínez-Bazan et al. 2007) is a graph database that efficiently stores and queries labeled and directed attributed multi-graphs. It keeps the graphs in secondary memory using different bitmaps. The graph model of DEX defines labeled nodes and directed edges where extra information is associated to each node and edge, rep-

[2] `http://sparsity-technologies.com/`

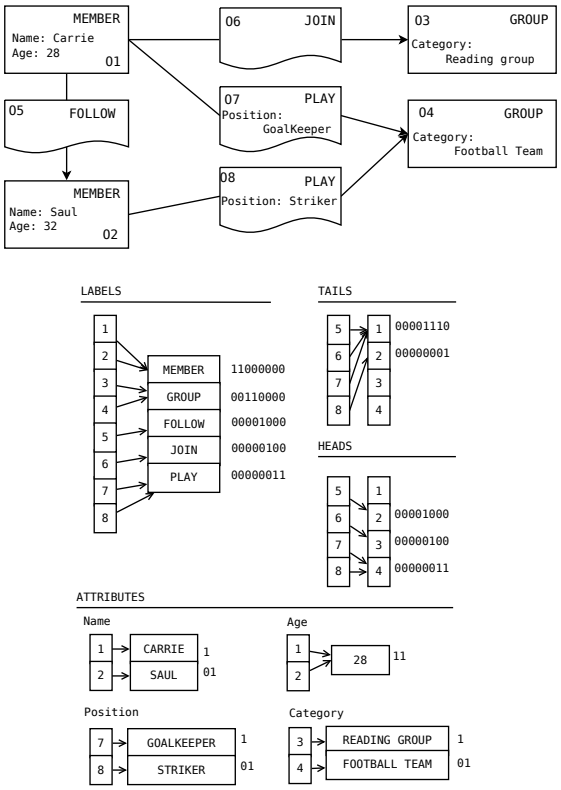

Fig. 1. Example of labeled attributed graph (top) and its DEX internal representation (bottom).

resented as a list of attributes. Therefore a graph in DEX is defined as $G = (V, E, L, T, H, \{A_1 \dots A_n\})$ where:

- V is the set of node keys.
- E is the set of edge keys.
- L is a key-value list that includes each key node (or key edge) and its label.
- T and H are key-value lists that associate each edge key to the keys of its corresponding source (tail, T) and target (head, H) nodes respectively.
- Each A_i in $\{A_1 \dots A_n\}$ represents a different attribute. Nodes and edges of the graph can take values for some of these attributes.

DEX represents this graph model through a set of bitmaps. Figure 1 shows an example of a graph internal representation in DEX. The top of the figure shows a graph for a social network where two members (*Carrie* and *Saul*) have joined different groups (a *reading group* and a *football team*). Users can be related to each other through a *follow* relationship, and users can participate in the groups through two different relationships (*join* for a *reading group* and *play* for a *football team*) described by different attributes (for example, the *position* attribute in the case of the *play* relationships). All the elements of the graph

(nodes and edges) have an associated object identifier. For instance, the object identifier of the member *Carrie* is *O1*.

The bottom part of the figure shows the bitmaps used in DEX to represent this graph. Labels storage is shown on the left. A map (implemented using a $B+$-tree) associates each object identifier (node or edge) to its corresponding label value. For instance, the object identifier *O3* is related to the label *Group* (since it represents the *Reading group*). Each label has an associated compressed bitmap that contains as many *bits* as objects in the graph. For a given label, a *one* in position i means that the object O_i is labeled with that label. However, in practice, bitmaps are only stored until the position of the last *one*. The $B+$-tree and these compressed bitmaps compose a double mapping. The purpose of storing this double mapping is to support bidirectional navigation. The first map is used to obtain the label of each object identifier. The bitmaps of each label answer the opposite query: given a label (for instance, *member*), obtaining the objects with that label is performed by retrieving the *ones* in the corresponding bitmap. The bitmap for *member* is [11000000], meaning that *O1* and *O2* are objects with the *member* label. The remainder information for the graph is managed in a similar way: a double map is used to represent the tails of the edges, another one represents the heads, and finally one map per attribute is stored.

The main purpose of this internal structure is to provide bidirectional access. That is, the data for a given node can be recovered using the maps indexed by its identifier. On the other hand, given a label or an attribute value, finding the nodes or edges with this label or value is performed by checking its corresponding bitmap and recovering the positions with a value *one*. Direct and reverse neighbor nodes are recovered by using the bitmaps *head* and *tail*.

DEX query engine is built over this internal representation. Its core implements a small set of primitives, and more complex queries are built on top of them.

2.2. Neo4j

Neo4j[4] is an open-source graph database that supports the storage and query of labeled directed attributed graphs. A graph in Neo4j can be defined as $G = (N, E)$, where N and E are the sets of nodes and edges respectively. Each node is a pair $n_i = (L_i, P_i)$, where L_i is the set of labels and P_i is the set of properties (or attributes) of the node. Labels of a node can be seen as tags. They can be used to define constraints over a group of nodes, to represent temporary states of nodes or, in general, to define a target group of nodes over which an operation will be performed. A property $p_j \in P_i$ is a key-value pair, where the value can be a primitive (the typical primitive types of any programming language like Boolean, Integer and String are supported) or a list of elements from one of these primitive types. An edge of a graph in Neo4j is defined as $e_i = (n_j, n_k, t_i, P_i)$, where n_j and n_k are the nodes linked through this edge, t_i is the label of the edge and P_i is the set of properties the edge contains (equivalent to the properties of the nodes). Neo4j defines its own query and update language, Cypher, which is a declarative language, where data is obtained by pattern matching.

Neo4j uses native structures for storing graph data, that is, it does not rely

[4] `http://neo4j.org/`

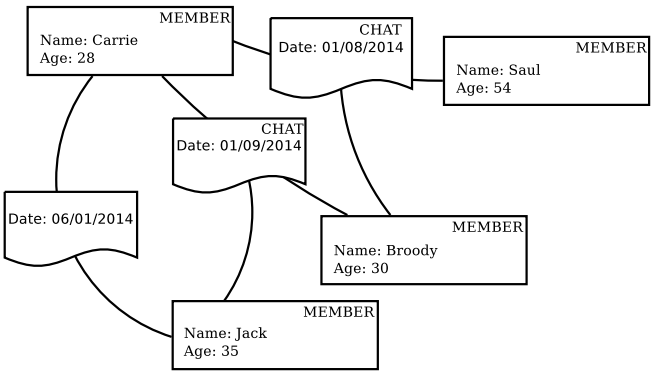

Fig. 2. An example of hypergraph model including n-ary edges

on traditional relational storage structures. As explained in (Robinson et al. 2013), the different parts of the graph, nodes, edges, and properties are stored in different store files. For example, nodes are stored in fixed-record files (with a length of 9 bytes), and the position of each node in the file is given by the node identifier. Similarly, edges are stored in fixed-record files (33 bytes per record, in this case). This file stores for each edge the identifiers of the source and target nodes, a pointer to the edge type, and pointers to the next and previous edges of the source and target nodes, which allow for a faster query processing. The properties of both nodes and edges are stored as key-value pairs in a property store.

2.3. HyperGraphDB

HyperGraphDB[6] (Iordanov 2010) is a graph database based on the Hypergraph model designed mostly for knowledge management, AI and semantic web projects. It supports a hypergraph $HG = (N, E)$, where N is the set of nodes and E is the set of edges, also known as links. The definition of links here is different from regular graphs, as links point to an arbitrary number of elements instead of just two, and links can be pointed to by other links as well. The hypergraph defines a more expressive structure that can be useful to model domains where more than two entities are usually related. For instance, each conversation of people in an online chat program could be modeled using this hypergraph model as a link that relates the participants in the conversation. Figure 2 shows an example of conversations along the time. For instance, the chat conversation in 01/08/2014 is represented as a link that involves 3 members.

HyperGraphDB uses an *atom* as the basic unit of representation. It contains a *typed value* and a *target set* composed of a set of atoms. Atoms can be nodes or links. A node is just an object value that does not point to anything else. Thus, the number of atoms in the target set of a node is 0. On the other hand, a link has at least one atom associated with it.

The storage of HyperGraphDB is basically distributed in two layers. The

[6] `http://hypergraphdb.org/`

primitive storage layer includes the information of the links (for each edge identifier the set of related identifiers is stored) and the data (the RAW value corresponding to each identifier). These data are stored in two associative arrays, implemented as key-value stores in BerkeleyDB. In addition to these two associative arrays, the system creates indexes on the data and allows the users to create additional indexes to speed up particular queries.

Over this primitive layer, the model layer manages the type system, the querying engine and some optimization facilities like caching and indexing. This layer manages the storage of the atoms, with their type, value and target set.

The HyperGraphDB query engine provides two different ways of specifying the query: an API to define standard graph traversals and a SQL-style language where a set of constraints over the required atoms are set.

2.4. SAP HANA Graph

SAP HANA[7] is an in-memory database management system developed by SAP. An important feature of HANA is that it includes a module for general graph database management, called HANA Graph (SAP 2016). It allows managing graph databases in which edges are directed, two given nodes can be connected by different edges, and both nodes and edges can have attributes consisting of an attribute name, a data type, and a value.

As described by SAP (2016), both nodes and edges must have an identifying attribute, called vertex key and edge key respectively, and edges must have two additional attributes, the source and target nodes they connect. Graphs in HANA are stored in two relations or views, one storing the nodes of the graph, and another one storing the edges. In addition to the identifying keys and the source and target attributes, any other attributes of nodes or edges will be stored as columns in the corresponding table. HANA Graph provides a SQL-based language called GraphScript, a procedural domain-specific language (Paradies et al. 2017) that allows the users to easily manage nodes and edges, and to implement typical graph processing algorithms.

2.5. SQLGraph

SQLGraph (Sun et al. 2015) is a proposal for property graph storage that relies on existing database management systems rather than on specific data structures for storing and processing the graph data. SQLGraph follows a hybrid approach that combines relational databases in combination with JSON stores. The adjacency information of the graph is stored in relations for the outgoing and incoming edges of the graph. However, this data is not represented plainly in the relations. Instead, hashing techniques presented in Bornea et al. (2013) are applied in order to improve query performance. The attributes of both nodes and edges are stored in additional JSON stores. In addition, these JSON stores keep a copy of the adjacency information of each edge, since this can improve the performance of certaing graph queries. The authors show in Sun et al. (2015) how queries expressed using Gremlin (Tinkerpop 2014), a procedural graph traversal

[7] `https://www.sap.com/products/hana.html`

language, can be translated into SQL queries on the relational database that stores the graph.

2.6. Plain relational representation of graphs

In addition to the systems we have reviewed in this section, attributed graph databases can also be created and managed in any relational database management system. A possible approach for this is analogous to the one used by HANA Graph, that is, storing the nodes and edges of the graph in two relations. The nodes relation would have a primary key and one column for each possible attribute present in any node. The edges table would have a primary key, mandatory source and target columns, and a column for each possible attribute present in an edge. Creating indexes on the primary keys of both relations, and on the source and target columns of the edges relation would improve the performance for basic graph operations, like obtaining the neighbors of a given node. Additional indexes created on certain attributes could also improve performance on queries involving those attributes. Another possibility for storing and processing graph databases in relational databases would be creating different relations for the different node or edge types. That is, if our graph contains three types of nodes, where all the nodes of the same type share the same attributes, we could create three relations, one for each node type. The same design would apply for edges. The advantage of this approach is that we would not have so many null values in the tables, although the query performance could be worse, depending on the structure of the graph and the specific queries of interest. In any case, although this is an option for representing attributed graph databases, the performance, both in terms of space and query times, would depend on the specific relational structure used to represent the graph.

2.7. Other systems

In the past years, many other graph database systems have emerged. They are focused on managing large amounts of data in a very efficient way. OrientDB[9] is a good example, which is document and graph oriented, implemented in Java and uses SQL as query language. In addition, many proposals were designed to work in distributed environments. Titan[10], Giraph[11], or Pregel (Malewicz et al. 2010) are just some examples.

3. Background: the k^2-tree

Before presenting our proposal in detail, in this section, we will briefly explain the data structure it is based on, that is, the k^2-tree (Brisaboa et al. 2014). The k^2-tree was originally proposed for compressing Web graphs, but can be used to represent any simple directed graph (that is, without attributes, labels, nor multiple edges linking two given nodes), and more generally, to represent

[9] http://www.orientechnologies.com/orientdb/
[10] http://thinkaurelius.github.io/titan/
[11] http://giraph.apache.org/

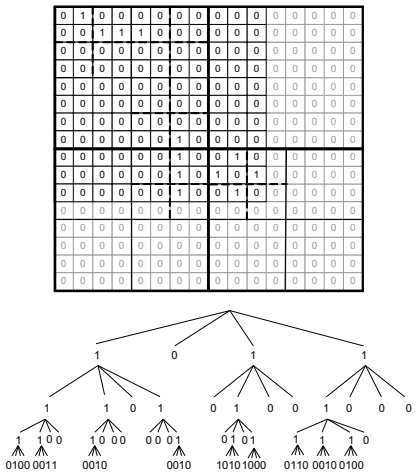

Fig. 3. Example of subdivision of an adjacency matrix (top) and resulting k^2-tree (bottom), for $k = 2$.

any binary relation. The k^2-tree is a compact tree structure created from the adjacency matrix of the graph by taking advantage of large empty areas in the matrix (that is, large areas in which there are no 1s). It achieves a very compact space representation of the graph, and supports efficient navigation, both forwards and backwards, without the need of decompressing. In addition, this representation supports some navigation possibilities not supported by other graph compression techniques, such as range queries over the adjacency matrix (that is, obtaining any submatrix), which are necessary in our proposal.

3.1. Data structure and construction

The k^2-tree is a tree-shaped representation of the adjacency matrix of the graph, built by recursively partitioning the adjacency matrix. The adjacency matrix of a graph with n nodes is a square matrix $\{A\}$ of size $n \times n$, where row i and column i correspond to the i^{th} node of the graph. The cell A_{ij} is 1 if there is a direct edge from node i to node j, and 0 otherwise. The k^2-tree obtains its best performance when there are large matrix areas containing only 0s, since these areas are represented with just one bit in the tree.

The construction of the k^2-tree consists in a recursive partition of the adjacency matrix following a MX-Quadtree strategy (Samet 2006). This partition is conceptually represented using a non-balanced k^2-ary tree, in which each node contains one bit, and k^2 children. Without loosing generality, let suppose that n is a power of k. In a first level, A is partitioned into k^2 submatrices of the same size. The root of the tree contains one child for each submatrix. The bit corresponding to each submatrix is 1 if the submatrix contains some 1, or 0 otherwise (that is, if the submatrix is an "empty" area of the adjacency matrix). Then, the partitioning procedure is recursively applied to each non-empty submatrix until we reach empty submatrices, or the cells of A containing a 1, which are

represented in the leaves of the tree. If n is not a power of k, the adjacency matrix can be artificially expanded so its size is the next power of k, filling the new cells with $0s$. Since the k^2-tree represents empty areas of the matrix with one bit, this expansion of the adjacency matrix will not affect the final result significantly.

Figure 3 shows an example of how the k^2-tree is built, with $k = 2$. In this example, the size of A is $n = 11$, but A has been expanded by adding 5 rows and columns, so the size of the new matrix is 16. In a first partition of A, only the up-right submatrix is empty and, therefore, represented with a 0. The rest of the submatrices are partitioned following the same schema, until we reach the leaves. The resulting tree is an alternative, more compact, representation of the adjacency matrix that allows us to obtain the value of a cell, row, column, or range of the adjacency matrix.

In order to avoid the use of pointers, the k^2-tree is represented in a very compact way using just two bit arrays: T (tree) and L (leaves). T stores all the bits of the k^2-tree except those in the last level. The bits are placed following a levelwise traversal: first the k^2 bits of the children of the root node, then the bits of the second level, and so on. L stores the last level of the tree. The k^2-tree uses an auxiliary structure that supports $rank$[13] operations over T in constant time, which will be required to navigate the compact representation of the tree, that is, to traverse down from the position of a node in T to the start position of its children.

3.2. Navigation

To find the direct (reverse) neighbors of a node in the graph, the k^2-tree needs to locate which cells in the row (column) of the adjacency matrix corresponding to that node have a 1. If we want to search for direct (reverse) neighbors in a k^2-tree, we go down through k children forming a row (column) inside the matrix, those submatrices that overlap with the row (column) of the node of the query. This top-down traverse can be performed efficiently over the compact representation of the tree, that is, the concatenation of the two bit arrays and the auxiliary structure. Detailed algorithms for these operations, as well as for range queries, are thoroughly described by Brisaboa et al. (2014).

While alternative compressed graph representations are limited to retrieving the direct, and sometimes the reverse, neighbors of a given node, the k^2-tree representation allows for more sophisticated forms of retrieval. First, in order to determine whether a given node u has a direct edge to a given node v, most compressed (and even some classical) graph representations have no choice but to extract all the neighbors of u (or a significant part of them) and see if v is in the set. The k^2-tree technique can answer such query in efficient time, by descending to exactly one child at each level of the tree. A second interesting operation is to find the direct neighbors of node u that are within a *range* of nodes $[v_1, v_2]$ (similarly, the reverse neighbors of v that are within a range $[u_1, u_2]$). Yet a third operation of interest is to find all the edges from a range of nodes $[u_1, u_2]$ to another $[v_1, v_2]$.

[13] $rank(B, i)$ computes the number of ones that are set up in bitmap B until position i.

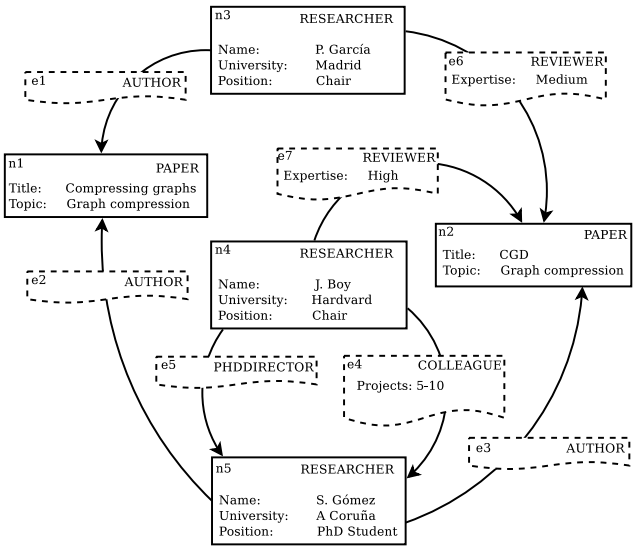

Fig. 4. Example of a labeled, directed, attributed multigraph.

4. Our proposal: AttK^2-tree

In this section, we propose our system, called *Attributed k^2-tree (AttK^2-tree)*, to efficiently store and process attributed graphs. The representation of the graphs is based on the k^2-tree, which, as seen, is a static data structure designed to work in main memory. Therefore, we propose an in-main-memory compact attributed graph representation designed to be used in contexts where big amounts of static data need to be intensively queried.

4.1. Graph model supported by AttK^2-tree

We described several attributed graph systems in Section 2. All of them are based on specific attributed graph models, presenting only small differences between them. Therefore, before describing the internal representation of our structure, we first consider the features of the attributed graph model the AttK^2-tree supports. Figure 4 shows an example of the attributed graphs supported by the AttK^2-tree. This graph represents a research network, including information about publications, authors, authorship, reviews, collaborations, and thesis supervision. Researchers and papers are modeled as nodes in the graph, and the different collaborations between them are reflected as edges (like thesis guidance and supervision, or collaboration in a research project). Researchers are related to the papers they *authored* through edges. Researchers can also be related with a paper through a *review* relation. This graph is an example of the data that could provide support to an application that detects conflicts of interest in order to assign reviewers to papers.

The graph model of the AttK^2-tree presents the following properties:

– Directed graph: the edges of the graph will be directed, meaning that they

distinguish between origin and target node. Figure 4 shows how edges from the graph explicitly identify their origin and target nodes. For instance, edge e_1 represents the authorship of a paper, having researcher n_3 as origin and paper n_1 as target of the edge. As usual, an edge of an undirected graph could be also represented in this model by using two directed edges (in opposed directions) between the two nodes it relates.

– Attributed graph: the attributes or properties are the most meaningful characteristic of general graphs. Many approaches can be followed to define the attributes of an element, including complex data types, range domains, and another constraints over each attribute. However, we define a more simplistic conception of attributes. Each node and each edge of the graph is described through a set of attribute-value pairs. Values are not restricted to a domain or a data type. They can take any value which will be managed as plain text. In Figure 4, we can observe that edge e_6 takes value *Medium* for attribute *Expertise*.
– Labeled graph: several definitions can be considered for a labeled graph. As it was described in Section 2, DEX considers that each component of the graph (nodes and edges) contains a unique label (or main value) that identifies the kind of element it belongs to. On the other hand, labels in Neo4j are considered as tags, supporting the definition of multiple tags for each element. We consider, in line with DEX, that each element of the graph (node or edge) has just one label, which we call *type*. This type determines the attributes that an element of that type can contain. In that sense, the label and the list of valid attributes for each label compose a *schema* which can be very helpful to work with domains with structured data. Figure 4 shows the label of each node and edge. In this example, two different labels are contemplated for the nodes. That is, that graph has only two node types: *Researcher* and *Paper*. The label determines the attributes that describe each node. Researchers are described through the attributes *Name*, *University* (where they work) and *Position* in that university. On the other hand, papers are described through the *Title* and the main *Topic* of the paper. Edges of the graph can be labeled with *Author*, *PhDDirector*, *Reviewer* and *Colleague*. *Colleague* relates two researchers that have collaborated in some research project. To summarize, labels are used to identify the type of a node or edge.
– Multigraph: AttK^2-tree does not constrain the number of edges connecting two different nodes, thus, many edges with the same origin and the same target can be defined. This characteristic is useful to represent in a natural way contexts where several kinds of relationships can be established between two nodes. Figure 4 shows an example of multigraph, where nodes n_4 and n_5 are related through two edges (e_4 and e_5) representing relationships with different nature (*PhDDirector* and *Colleague*) between those nodes. This multi-edge nature, combined with the labeled and attributed properties, makes this model very expressive. Therefore, AttK^2-tree can fit with the structural properties of many real graphs.

4.1.1. Formal definition

A formal definition of a labeled, directed, and attributed, multi graph *(LDAM)* is represented as a 10-tuple $G = (L_N, L_E, N, E, R, L_A, S_N, S_E, A_N, A_E)$, where:

– L_N is the set of possible labels that the nodes of the graph can take. For the

Table 1. Nodes schema.

Label (l_i)	Attributes($\{a_j\}$)
Paper	$\{Title, Topic\}$
Researcher	$\{Name, University, Position\}$

graph in Figure 4, $L_N = \{Paper, Researcher\}$, so there are two node types in this example.

- L_E is the set of possible labels that the edges of the graph can take. Regarding to the same example, $L_E = \{Author, Colleague, PhDDirector, Reviewer\}$, so there are four types of edge.
- $N = \{(n_i, l_j)\}$ is the set of nodes, being $n_i \in 1 \dots |N|$ a numeric identifier of the node and $l_j \in L_N$ the label of the node. In the example, the set of nodes is composed of

$$N = \{(1, Paper), (2, Paper), (3, Researcher), (4, Researcher), (5, Researcher)\}.$$

- $E = \{(e_i, l_j)\}$ is the set of edges, where $e_i \in 1 \dots |E|$ is the identifier of the edge and $l_j \in L_E$ is its label. The set of edges of the graph in Figure 4 is

$$E = \{(1, Author), (2, Author), (3, Author), (4, Colleague),$$
$$(5, PhDDirector), (6, Reviewer), (7, Reviewer)\}.$$

- R contains the relations between the nodes, that is, the origin and target nodes of the edges of the graph. Each element of R is a triple (e_i, o_i, t_i), where e_i is the edge identifier, o_i is the identifier of the origin node of the edge, and t_i is the identifier of the target node of the edge. It is easy to note that, by definition, $|R| = |E|$. In the example:

$$R = \{(1, 3, 1), (2, 5, 1), (3, 5, 2), (4, 4, 5), (5, 4, 5), (6, 3, 2), (7, 4, 2)\}.$$

- $L_A = \{a_i\}$ is the set of the different attribute labels of the graph. In other words, L_A is the union of all different attributes that describe the nodes and the edges of the graph. In the example,

$$L_A = \{Title, Topic, Name, University, Position, Projects, Expertise\}.$$

- $S_N = \{sn_i\}$ is the set of schemas for the types of the nodes. Each element of S_N defines the set of attributes that can be used for a given node type. Each element sn_i of the schema is represented as a pair $(l_i, \{a_j\})$ that associates a set of attributes $\{a_j\} \subseteq L_A$ to a given node type $l_i \in L_N$, where the node label $l_i \in L_N$ has associated a set of attributes $a_j \in L_A$ defined by that node type. Note that an attribute is not exclusive of a node type. In other words, several node types can be described through the same attribute. Table 1 shows the schema of the nodes for the graph in Figure 4.
- $S_E = \{se_i\}$ is the set of schemas for the types of the edges, in a completely analogous way to S_N. Each element of S_E defines a valid schema for an edge type. Each $se_i = (l_i, \{a_j\})$ is a pair where $l_i \in L_E$ is the corresponding edge label and $\{a_j\}$ is the set of valid attributes for each edge type. Table 2 shows the schema of the nodes for the graph in Figure 4.
- $N_A = \{(n_i, a_j, v_k)\}$ defines the properties of the nodes. It is a set of triples, where each triple defines the value v_k that the node $n_i \in 1 \dots |N|$ takes for

Table 2. Edges schema.

Label (l_i)	Attributes ($\{a_j\}$)
Author	{ }
Colleague	{*Projects*}
PhDDirector	{ }
Reviewer	{*Expertise*}

Table 3. Attributes for node n_3.

Node Identifier (n_i)	Attribute (a_j)	Value (v_k)
3	*Name*	*P. García*
3	*University*	*Madrid*
3	*Position*	*Lecturer*

the attribute $a_j \in L_A$. Note that a triple (n_i, a_j, v_k) is valid in a data source if $\exists\ l_m | (n_i, l_m) \in N \wedge (l_m, \{\dots a_j \dots\}) \in S_N$. That is, a node can only take a value for an attribute included in its schema, given by its node type. For instance, the set of triples describing the properties of node n_3 in Figure 4 are shown in Table 3.

– $E_A = \{(e_i, a_j, v_k)\}$ describes the properties of the edges (analogously to N_A). As an example, the triple describing edge e_6 is provided in Table 2.

Next, we detail the internal representation of AttK^2-tree designed to support the graph model presented in this section.

4.2. Data structure

The AttK^2-tree stores a directed, attributed, and labeled multi-graph by using binary relations represented with k^2-tree structures. It is a compressed solution composed of a set of k^2-trees and some additional data structures. The AttK^2-tree represents the graph with three components: the schema of the data, the data included in the nodes and the edges and, finally, the relations between the elements of the graph topology. Next, we present the three components of the AttK^2-tree.

4.2.1. Schema

The schema of the graph comprises the set of valid node labels (types) and edge labels (types), and the valid attributes for each of them, that is, the attributes that can be used for each node or edge type. The schema component works as

Table 4. Attributes for edge e_6.

Edge Identifier (e_i)	Attribute (a_j)	Value (v_k)
6	*Expertise*	*Medium*

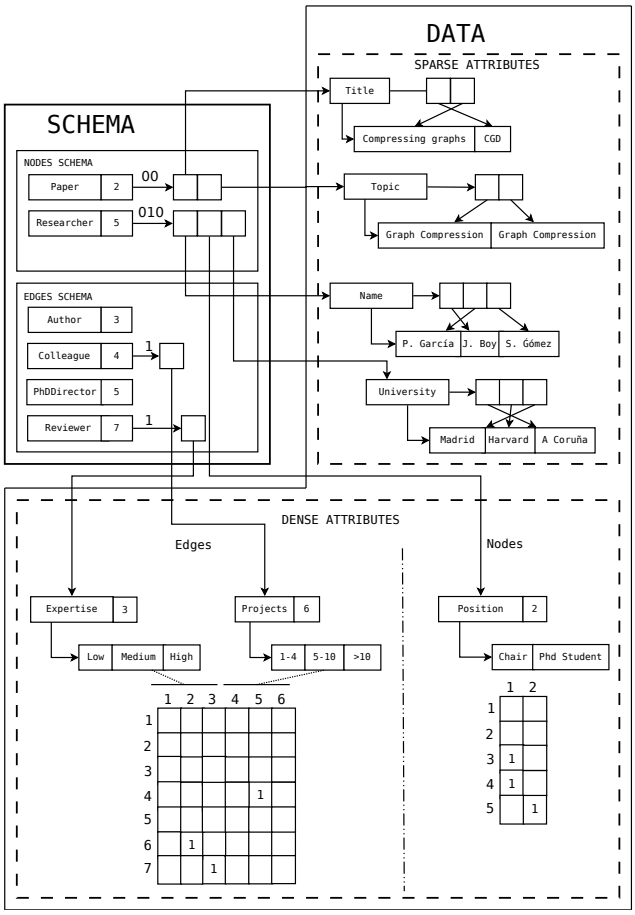

Fig. 5. Internal representation of Schema and Data subsystems in AttK^2-tree.

an index for the other components of the AttK^2-tree. The elements of the graph model L_N, L_E, N, E, S_N, S_E are stored in this schema layer. Figure 5 on the left shows the schema storage for the graph of the example. It is composed of:

– **Nodes schema**: the nodes schema keeps the valid node labels, and the label each node of the graph has. To keep this part of the graph compact, node labels are sorted lexicographically, and node identifiers are assigned sequentially to the nodes of each label. That is, the first m_1 nodes with the first type in the schema will have identifiers from 1 to m_1. The m_2 nodes from the second node type will have a range of identifiers from $m_1 + 1$ to $m_1 + m_2$ and so on. Each entry of the nodes schema will store the highest node identifier with this label. Figure 5 shows the nodes schema for the graph at Figure 4. It has two entries: *Paper* (having 2 as the highest identifier) and *Researcher* (with limit 5). That means that the nodes with identifiers in the range from 1 to 2 are papers, while the nodes with identifiers from 3 to 5 are researchers. Each label also points to its valid attributes in the data subsystem, and includes a bit array

indicating whether each of its attribute is dense or sparse (further explained in the next subsection).

– **Edges schema**: a table storing the edges schema is implemented in the same way as the nodes schema. Therefore, the edge identifiers will also be ordered by type. In that way, given an edge identifier, its corresponding type can be computed by performing a binary search over the entries in the schema. For instance, to recover the type of edge 6 in Figure 5, a binary search over the upper limits of each edge label is performed, until reaching the range 5–7 that includes it, concluding that the node type of edge 6 is *Reviewer*.

The schema layer is the starting point of the internal representation of the graph in the AttK^2-tree, providing indexed access to the other two layers. It is used to retrieve the ranges of identifiers for a label, and to recover the label for a given identifier. It also stores references to the valid properties for a given label.

4.2.2. Data

Part of the data component of the graph contains the attribute values for the nodes and edges of the graph. It stores the values that each element of the graph (node or edge) takes for each valid attribute according to its type and the schema of the graph. Each different attribute can be represented in two different ways depending on the frequency distribution of its values. One type corresponds to the *dense attributes*, where many nodes or edges of the graph share the same value for that attribute. In opposition to the dense attributes, in *sparse attributes* nodes or edges usually take different values for that attribute. Titles, URLs, or identifiers are common examples of sparse attributes whereas age or nationality are examples of dense attributes. These two types of attributes will have a different internal representation in the AttK^2-tree:

– **Sparse attributes**: attributes for which the graph elements (nodes and edges) usually take different values will be stored as a list indexed by element identifier. This list is double-indexed: in addition to the implicit index by element identifier, there is an additional index to maintain the entries in lexicographical ordering. This additional index is used to recover the elements taking a specific value by a binary search. Figure 5 (top-right) shows four sparse attributes: *Title, Topic, Name, University*. For instance, *Name* is a valid attribute for the node type *Researcher*. The values in the list are sorted by node identifier. The first element of this list *(P. García)* is the value that the first researcher (n_3) takes for attribute *Name*. Given a node (n_i, l_j), its value for a sparse attribute will be in the position $i - limit + 1$, where *limit* represents the lowest node identifier of the type l_j. The additional index provides support to perform a binary search over the attribute values. We can see in the example that the first element of this additional index in the attribute *Name* points to *J. Boy*, the first element of the list in a lexicographical order.
– **Dense attributes**: all dense attributes of the graph are stored in two k^2-trees: a k^2-tree for the dense attributes of the nodes, and another k^2-tree for the dense attributes of the edges. The k^2-tree for the dense edge attributes is built as follows (the k^2-tree for the dense node attributes is built in the same way): each dense attribute A_i can be seen as a binary relationship between the $|E|$ edges and the set of different values that those edges take for that attribute. These relationships can be represented in consecutive columns of the adjacency

matrix. Rows of the adjacency matrix represent the edges of the graph, ordered by their identifiers. Columns will represent the possible different values of each attribute. Each group of consecutive columns represents the different values for an attribute. A 1 in a cell (i, j) of this adjacency matrix means that the edge with identifier i takes the value j for the attribute located in this range of columns. This adjacency matrix is represented using a k^2-tree. An additional structure stores, for each attribute, the block of columns that correspond to this attribute, storing again the upper limit for each attribute, and the specific values that represent each column.
Figure 5 (bottom-right) shows the representation of the dense attributes. The adjacency matrix for the nodes is on the right, where the 5 rows represent the 5 nodes of the graph. The adjacency matrix on the left contains a row for each one of the 7 edges. On the top of this adjacency matrix, the meaning of each column is specified by several lists. The attribute *Expertise* contains three possible values *(Low, Medium, High)*. This attribute includes the index 3, which indicates that its three columns end at column 3 of the global adjacency matrix. Then, the cell $(6, 2)$, which contains a 1, means that the edge e_6 takes the value *Medium* for attribute *Expertise*. On the other hand, attribute *Projects*, which is specified in the Schema as a valid attribute for the edge type *Colleague*, contains three possible values which end at column 6 of the global adjacency matrix. In that way, the 1 in cell $(4, 5)$ means that e_4 takes value *5–10* for attribute *Projects*. Note that in some regions of this matrix, due to the schema constraints, no ones can appear. For instance, the matrix between the rows 1–3 and the columns 1–3 is empty because label *Author* does not have attribute *Expertise*, according to the schema.

Note that for some attributes, the choice of representing them as a sparse or dense attribute could be not obvious. A possible criteria could be based on the number of different values regarding to the number of elements taking a value for that attribute.

To indicate if an attribute is dense or sparse, AttK^2-tree stores in its schema a bit array for each node/edge type, containing as many bits as valid attributes it has. A 0 value indicates that the attribute is sparse, and a 1 value indicates that the attribute is dense. For instance, the bit array corresponding to node label *Researcher* is 010, which indicates that the first attribute (*Name*) is sparse, the second attribute (*Projects*) is dense, and the third attribute (*University*) is sparse.

4.2.3. Relations

The third component of the AttK^2-tree stores the *Relations*, that is, the different edges that connect the nodes of the graph. We store these relations with a k^2-tree, which needs to be extended to store the edge identifiers corresponding to each connection.

The k^2-tree represents, in a very compact way, simple graphs that can be represented by an adjacency matrix. A *one* in cell (i, j) shows the existence of an edge from the node i pointing to the node j. However, additional information is needed to store the relations in AttK^2-tree. First of all, each *one* of the matrix has to be related to its edge identifier, which is used as pointer to the data layer (for instance, to recover the attributes of that connection). On the other hand, the AttK^2-tree supports multi-graphs. This means that more than one edge can

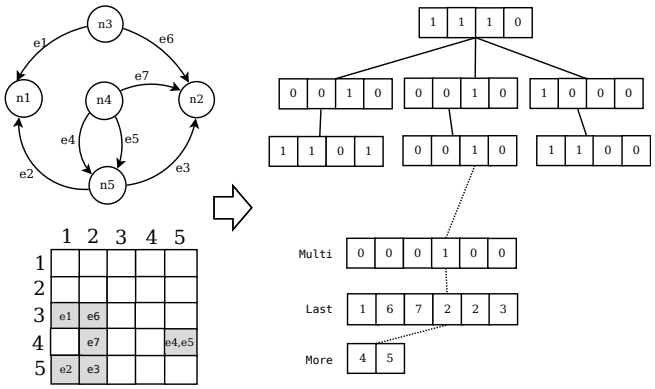

Fig. 6. Relations component in the AttK^2-tree.

relate a pair of nodes. So, several edges can be represented in the same cell of the adjacency matrix. Figure 6 shows the relationships of the example and the corresponding adjacency matrix containing those edge identifiers. For instance, cell $(4, 5)$ contains two edge identifiers because two different edges connect n_4 and n_5 in the original graph (e_4, e_5).

The relationships in AttK^2-tree are stored with the original k^2-tree and some additional data structures to represent multi-edges and to trace their edge identifiers. Figure 6 shows the structure we call multi-edge k^2-tree, which is composed of the following elements:

- **k^2-tree**: a k^2-tree is built to represent a binary relation among nodes in which two nodes are related if at least one edge connects them in the original graph. It is a standard k^2-tree except for the fact that in this case, the bitmap of the last level also needs an additional structure to perform rank operations. By allowing rank operations, it is possible to compute the relative position of a bit with value 1 in the last level of the tree, so it can be used as an index to the *Multi* bitmap (which will be explained in the next paragraph). For instance, the last level of the tree in Figure 6 contains 6 *ones*. If we perform a rank operation over the bitmap until the 7-th position, we have that this is the fourth leaf of the k^2-tree, and we can check the fourth value of *Multi* bitmap to check whether this leaf represents one or several edges connecting the same pair of nodes.
- **Multi**: is a bitmap that stores, for each *one* element of the leaf level, whether it is a multiple edge or it is representing only one edge. Therefore, *Multi*$[i]$ will have value *one* if the i-th one of the k^2-tree is clustering multiple edges. In the example, only the fourth position of the bitmap *Multi* contains a *one* value (clustering edges e_4 and e_5). This information is used to read the next array (*Last*).
- **Last**: this array stores the last edge identifier of each *one* of the k^2-tree. For the i-th one of the leaf level, if it is a single edge (that is, if *Multi*$[i] = 0$) then *Last*$[i]$ contains the identifier of that unique edge. Otherwise, when the i-th one is a multiple edge, *Last*$[i]$ represents the position in *More* array, where the last edge is located. In the example of the Figure 6, *Multi*$[1] = 0 \wedge$ *Last*$[1] = 1$, so the last (and unique) edge (corresponding to cell $(3, 1)$) has identifier 1. On the other hand, *Multi*$[4] = 1 \wedge$ *Last*$[4] = 2$ so the last edge of cell $(4, 5)$ will

be in *More*[1]. It is possible to obtain the position of the first edge in array *More*, as it will be the following to the last edge of the previous multi-edge. Thus, we first need to compute where the previous multiedge is in bitmap *Multi*, which can be denoted p, and then access to $Last[p+1]$. Position p is computed using rank and select operations[18] over *Multi*, more concretely, $p = select(rank(Multi, i) - 1)$. Thus, in case that $Multi[i] = 1$, edge identifiers are located in $More[b], \ldots, More[e]$, with $b = Last[select(rank(Multi, i) - 1) + 1]$ and $e = Last[i]$.

– **More**: This array contains the identifiers of the multi-edges and it is indexed by the *Last* and *Multi* arrays. Figure 6 shows the two identifiers for the only multi-edge of the example: e_4 and e_5, corresponding with cell $(4, 5)$.

The three layers Schema, Data and Relations compose the internal representation of AttK^2-tree, used to store directed, attributed, labeled multi-graphs. These structures, based on the usage of k^2-trees, were designed to provide a compressed representation of attributed graphs, which could be accessed by basic queries. The next section presents the navigation over the internal representation of AttK^2-tree.

5. Navigation and operations

In this section we present the query API of AttK^2-tree composed by a set of basic operations over attributed graphs. This API aims to provide a basis for the construction of more complex queries. Our API contains 12 operations, which can be classified according to the layer of AttK^2-tree they imply.

5.1. Operations over the schema

Some of the basic operations in the AttK^2-tree work with the labels (types) of the graph.

– **Retrieval of labels**. The operation *Get*{*Node*|*Edge*}*Types* returns the different labels of the nodes or the edges of the graph. According to the internal representation of the AttK^2-tree, it is trivially implemented by recovering all entries of the Nodes Schema (or the Edges Schema). In the graph of the example, *GetNodeTypes* returns the labels *Paper* and *Researcher*. On the other hand, *GetEdgeTypes* returns the labels *Author, Reviewer, PhDDirector, Colleague*.

– **Filter by type**. *Scan*{*Nodes*|*Edges*}(*Type*) returns the nodes or the edges of a given type. Taking into account that the identifiers were allocated according to the type of the elements, this operation becomes quite straightforward. For instance, the operation *ScanNodes*("*Researcher*") is implemented by performing a binary search over the labels in the Nodes Schema, showed in Figure 5. When the entry 2 is retrieved, the upper limit of the range of identifiers with label *Researcher* is obtained, with value 5. The lower limit of the range is retrieved from the previous entry (that is, the first entry) with value 2. Therefore, *Researcher* nodes range from 3 to 5.

[18] $select(B, i)$ obtains the position in B of the i-th 1.

– **Find by element identifier**. *Get*{*Node*|*Edge*}*Type*(*id)* gets the type corresponding with an identifier. *GetNodeType* starts by performing a binary search over the upper-limits of the Node Schema, until the correct range is found. Then the label can be returned. For instance, *GetNodeType*(4) starts from a binary search over the nodes schema, until the lowest upper-limit is found (in this case is the second entry with value 5) and the highest lower limit (the first entry with value 2). Consequently, the node 4 has type *Researcher*. The behavior of *GetEdgeType*(*id*) is totally symmetric to *GetNodeType*(*id*) and it is implemented exactly in the same way.

5.2. Operations over the Data

Next four operations involve the data subsystem. They work over the attribute values of the nodes and edges of the graph.

– **Attribute retrieval**. *Get*{*Node*|*Edge*}*Attribute*(*id*, *att*) is the basic operation that obtains the value that a node (or edge) with identifier *id* takes for the attribute with label *att*. The operation starts by obtaining the type of the given node, which is solved with the operation *GetNodeType*(*id*). Then, the list of valid attributes of the node is checked looking for label *att*. If label *att* is not included in the list of valid attributes for that type, then the attribute is undefined for that node and no result is returned. For instance, *GetNodeAttribute*(3, "*Title*") in the example searches the list of attributes of type *Researcher*, which are *Name*, *University*, and *Position*; thus, *Title* is not a valid attribute and no result is returned. Otherwise, the attribute is checked. By accessing to the label bitmap it is possible to determine if the attribute is sparse or dense. If it is a sparse attribute, the procedure is quite simple: the list of plain values is checked at position $id - limit + 1$, where *limit* is the lowest identifier of the type *GetNodeType*(*id*). The value of this cell is returned. For instance, for *GetNodeAttribute*(3, "*Name*") it obtains that *Name* is the first label of node type *Researcher*, thus, it checks the first position of its bitmap. It is a 0, thus, it is a sparse attribute. Then it will return the value of the position 2 in the list of values for *Name*, that is, *J. Boy*. For dense attributes, a range operation has to be performed in the k^2-tree. The range includes only one row (corresponding to *id*) and the columns representing the dense attribute that is being checked. For instance, for the operation *GetEdgeAttribute*(6, "*Expertise*") it obtains that *Expertise* is the first label of edge type *Reviewer*, thus, it checks the first position of its bitmap. It is a 1, thus, it is a dense attribute. Then, the row 6 between the columns from 1 to 3 is checked. A *one* appears in the second column, so the value corresponding to the second position in the *att* list, *Medium*, is finally returned as a value.
– **Filter by attribute value**. *Select*{*Nodes*|*Edges*}(*Type*, *att*, *val*) returns all nodes (or edges) belonging to *Type* that take value *val* for attribute *att*. It is a classical filtering by property and type. In the graph of the example, queries like *researchers from Coruña, papers with the topic Graph Compression* or *PhD Students* are examples of this select operation. The operation starts by recovering the entry corresponding to the specified *Type* in the same way *Scan*{*Nodes*|*Edges*}(*Type*) does. It obtains the lower and upper limits for the identifiers of that type, (*lower*, *upper*), which will be necessary later for this query. Then, attribute *att* is searched in the attribute list of that entry.

If it is a dense attribute, the value is searched in the list of labels of that attribute in order to compute the limits of the needed range search over the k^2-tree. For instance, when *SelectNodes*("*Researcher*", "*Position*", "*Chair*") is queried, a range query is performed between rows from 4 to 5 (since these are the lower and upper limits of the *Researcher* type) and column 1 (corresponding to value *Chair*). The rows taking a *one* in this range will be returned as a result (in this case, nodes 3 and 4). Since attributes are located in the k^2-tree ordered by value, in addition to the equality, other patterns of comparison could be implemented efficiently. For sparse queries, the operation is similar (binary search over all the labels of this attribute list). When the valid values are reached, their positions determine the node identifiers which have to be returned.

5.3. Operations over the Relationships

The last kinds of queries involve conditions over the relationships of the graph. Two basic queries can be the basis of the exploration of the relationships in the graph:

- **Find neighbors by node type**. *Neighbors*(*Type*, *id*) returns all nodes of the specified type that are neighbors of the node with the identifier *id*. The operation starts by retrieving the range of valid identifiers (*lower*, *upper*) according to the given type. Then the multi-edge k^2-tree is explored in row *id* and between columns (*lower*, *upper*). Consider the query *Neighbors*("*Researcher*", 4), asking for the neighbors of node 4 (*researcher J. Boy*) which have *Researcher* type. First of all, the limits of *Researcher* are computed, obtaining $(3, 5)$. Therefore, a range query between row 4 and columns $(3, 5)$ is performed. A multi-edge in the cell $(4, 5)$ is recovered (containing edges e_4 and e_5); thus, the node 5 (*researcher S. Gómez*) is the result of that query.
- **Find neighbors by edge type**. *Related*(*Type*, *id*) returns all nodes related to the identifier *id* connected to them through an edge with the given *Type*. In this operation, the filtering is in the edge identifier, which is recovered after performing the query over the k^2-tree. Hence, this filtering has to be processed after finishing the query. The query is executed as follows. First of all, the valid range of identifiers of the given edge type is computed. Therefore, the full row *id* is queried in the multi-edge k^2-tree. After that, all the results are processed sequentially, removing from the result the columns that do not contain any edge in the range of edge identifiers for the given edge type. The result will be the identifiers of the remaining columns. For instance, the query *Related*("*Author*", 3) starts by computing the valid identifiers for *Author*, which are 1–3. Then, the row 3 is queried in the multi-edge k^2-tree, obtaining two cells with results: $(3, 1)$ and $(3, 2)$. The edge identifier of the cell $(3, 1)$, that is e_1, is included in the range of valid identifiers, so n_1 is a result of that query (the paper *Compressing graphs*). However, the edge identifier in cell $(3, 2)$ is not valid (e_6 has type *Reviewer*) so node n_2 is not returned as a result.

The set of operations we implement in AttK^2-tree aims to provide a basic but efficient querying to the attributed graphs. More complex queries can be implemented on the top of these basic operations as intersections, unions or chains

of them. For instance, the query *Papers reviewed by P. García and written by S. Gómez* could be implemented as an intersection of three different operations:

– *ScanNodes("Paper")*
– *Related(SelectNodes("Reviewer", "Researcher", "Name", "P. García"))*, and
– *Related(SelectNodes("Author", "Researcher", "Name", "S. Gómez"))*.

The experimental evaluation in Section 7 gives some experimental results of the spatial requirements and the temporal efficiency obtained by AttK^2-tree. Furthermore, as a proof of concept, it is evaluated with other proposals in the state of the art.

6. Dynamic AttK^2-tree

The approach described in the previous section is static, thus, it requires knowing in advance the whole graph database we want to store. However, this may be a limitation for some possible application scenarios. In this section, we describe the dynamic AttK^2-tree, denoted dynAttK^2-tree, which allows changes in schema, data, and relationships.

6.1. Data structure

The dynAttK^2-tree can also store a directed, attributed, and labeled multigraph. Its dynamic nature relies on several dynamic data structures (Navarro 2016, Chapter 12), especially dynamic k^2-trees (Brisaboa et al. 2017), dynamic wavelet trees (Mäkinen and Navarro 2008), dynamic bit arrays and dynamic vectors. Analogously to AttK^2-tree, dynAttK^2-tree represents the graph with three components: the schema of the data, the data included in the nodes and the edges and, finally, the relations between the elements of the graph topology. Next, we present these three components.

6.1.1. Schema

Whereas the static AttK^2-tree reorders nodes and edges by type, such that by knowing their identifier, their type can be efficiently computed (by means of a binary search), this cannot be done in the dynamic variant. Node and edge identifiers are given in order as they are inserted into the database. Thus, we use two dynamic sequences to represent the node/edge types, sorted by their identifier. More concretely:

– **Nodes schema**: the nodes schema keeps the valid node labels, and the label each node of the graph has. To keep this part compact and dynamic, node labels are stored using dynamic vectors where the labels are sorted lexicographically. In addition, the type of each node is stored using a dynamic sequence, more concretely using dynamic wavelet trees, which allow us to efficiently locate all nodes of a given type and also obtain the type of a given node, using little space[21]. For each node label, we include a vector of attributes and a dynamic bit array, indicating, for each attribute, if it is a sparse or dense.

[21] A wavelet tree is a data structure that maintains a sequence S of n symbols supporting

– **Edges schema**: the edges schema is implemented in the same way as the nodes schema, storing the edge types sorted lexicographically in a dynamic vector. Again, dynamic wavelet tree is used for storing the type of each edge in compact space while still allowing efficient searches. Again, for each edge label, we include a vector of attributes and a dynamic bit array, indicating, for each attribute, if it is a sparse or dense.

As with the static version, the schema layer is the starting point of the internal representation of the graph in the dynAttK^2-tree, providing indexed access to the other two layers. It is used to retrieve the different labels of node and edges, and to recover the label for a given identifier. It also stores references to the valid properties for a given label. In contrast to the static version, the proposed dynamic variant allows changes on the node or edge schema. For instance, it is possible to include new node/edge types or new attributes for a node/edge type.

6.1.2. Data

We also differentiate among sparse and dense attributes:

– **Sparse attributes**: are represented analogously to the sparse attributes in the static version, but using dynamic lists.
– **Dense attributes**: are represented slightly different compared to the static version. AttK^2-tree uses only two k^2-trees, one for the dense nodes attributes and another for the the dense edges attributes. We allow changes in the data schema, and more particularly, new value attributes can appear for dense attributes. Since the dynamic k^2-tree does not support efficiently adding columns or rows in the middle of the matrix, but at the end, it becomes more convenient to use one different dynamic k^2-tree for each dense attribute, such that new attribute values are always appended at the end of its attribute matrix.

6.1.3. Relations

The third component of dynAttK^2-tree follows the same approach as the static AttK^2-tree. It uses a dynamic k^2-tree for storing the relations among nodes and a dynamic bit array to store bitmap *Multi*. The main difference is found in the representation of the list of multiedges for a given pair of nodes. In the dynamic variant, we do not use arrays *Last* and *More*, but a dynamic vector containing, for each leaf of the tree, a dynamic vector of edge identifiers.

6.2. Operations and Navigation

Navigation is done analogously to that of the static AttK^2-tree, described in Section 5. The main substantial difference appears for the operations over the schema *Get*{*Node*|*Edge*}*Type*(*id*), as labels do not depend on their identifier, but must be queried over a wavelet tree.

---

the following operations: $access(S, i)$, which returns the symbol at position i in S; $rank_c(S, i)$, which counts the times symbol c appears up to position i in S; and $select_c(S, j)$, which returns the position in S of the j-th appearance of symbol c. They can be efficiently implemented using compressed space and perform well in practice. Wavelet trees and their applications have been extensively described by Navarro (2014).

Moreover, as this dynamic version allows changes in the schema, the data, and the relationships (insertions, deletions and updates), new operations appear to support these functionalities. Basically, these operations rely on the dynamism of the underlying structures (insertions, deletions and updates over the dynamic k^2-trees, dynamic wavelet trees, dynamic bit arrays and dynamic vectors).

7. Experimental evaluation

In this section, we analyze the spatial and temporal performance of our structure, which was designed to support basic operations over an attributed graph in a very compact way. We compare our structure with DEX, Neo4j, and HANA Graph, three of the most relevant graph database management systems in the state of the art. However, it is important to note that the results are provided in order to prove that we propose a compact structure with some basic search capabilities that is competitive in terms of space and time, but we are not proposing an alternative to these systems, since the purposes of our structure are different. We designed a compact attributed graph representation with some queryable capabilities and we implemented some basic operations, but our structure is not a full graph database platform. Neither AttK^2-tree nor dynAttK^2-tree support all the algorithms and operations that are characteristic in those kinds of engines. Thus, this comparison has to be understood just as a proof of concept of the structure we propose.

7.1. Experimental Framework

We ran experiments on a dedicated Intel® Core™ i7-8700K CPU @ 3.70GHz (12 cores) with 12MB of cache, and 64GB of RAM. It ran Ubuntu 16.04.1 LTS with kernel 4.4.0-31 (64 bits).

7.1.1. Tools

We include here some specific details on how each system was configured for launching the queries.

- AttK^2-tree: was implemented in C, compiled with gcc (version 5.4.0).
- dynAttK^2-tree: was implemented in C++, compiled with gcc (version 5.4.0). It uses *DYNAMIC*[23], a succinct and compressed dynamic data structures library implemented by Nicola Prezza (Prezza 2017).
- DEX: corresponds to the very compact graph database described in Section 2.1. We implemented the operations that AttK^2-tree supports through a Java program that invokes the corresponding native functions of the DEX library.
- Neo4j: is the commercial graph database system described in Section 2.2. In order to execute the same operations implemented over AttK^2-tree, the queries were implemented in Cypher language. Those Cypher queries are called from a program implemented in Java, which uses the Neo4j Java driver.

[23] `https://github.com/xxsds/DYNAMIC`

– HANA Graph: is the commercial graph database system described in Section 2.4. More concretely, we have used the official SAP HANA, express edition, which is available for free. We implemented the queries using a program implemented in Java and connecting to the database using a JDBC driver, more concretely using the ngdbc library (sap.jdbc).

7.1.2. Queries

We measured the performance of our structure through the execution of 8 different kinds of queries, whose implementation in our structure was described in Section 5. They include operations over the schema, the relations and the attributes of the graphs.

We designed a synthetic query set of 1,000 queries per each kind of operation:

– **Query set 1**: *GetNodeType* obtains the type of a given node.
– **Query set 2**: *GetEdgeType* obtains the type of a given edge.
– **Query set 3**: *GetNodeAttribute* obtains the value that a given node takes in a specific attribute.
– **Query set 4**: *GetEdgeAttribute* obtains the value that a given edge takes in a specific attribute.
– **Query set 5**: *SelectNode* obtains the set of nodes that takes a given value for an attribute.
– **Query set 6**: *SelectEdge* obtains the set of edges that takes a given value for an attribute.
– **Query set 7**: *Neighbors* returns the nodes of a given type related to a node.
– **Query set 8**: *Related* returns the nodes related to a given one through a specific edge type.

Note that the previously described operations *ScanNodes* and *ScanEdges* are not included in this evaluation as they are relevant only for our proposed structure. We have not included *GetNodeTypes* and *GetEdgeTypes*either, as they can be executed very fast and they lack of interest in this comparison.

These query sets are analyzed in three categories. On one hand, the operations over the schema (queries 1 and 2), which are the most simple queries. The times obtained by each alternative for these operations give a brief idea of the minimal time of communication with the database. Queries from 3 to 6 represent operations over the data (properties and types) of nodes and edges. Finally, query sets 7 and 8 establish conditions over the relationships in the graph.

7.1.3. Datasets

Movielens 100k (`ML100k`) The first use case we analyze is a dataset extracted from a movie recommendation website, Movielens[24], which contains ratings of movies from different users of the web, including statistical information of the users and tags of the movies. We use a subset of 100,000 ratings for 1,682 movies

[24] http://movielens.umn.edu/

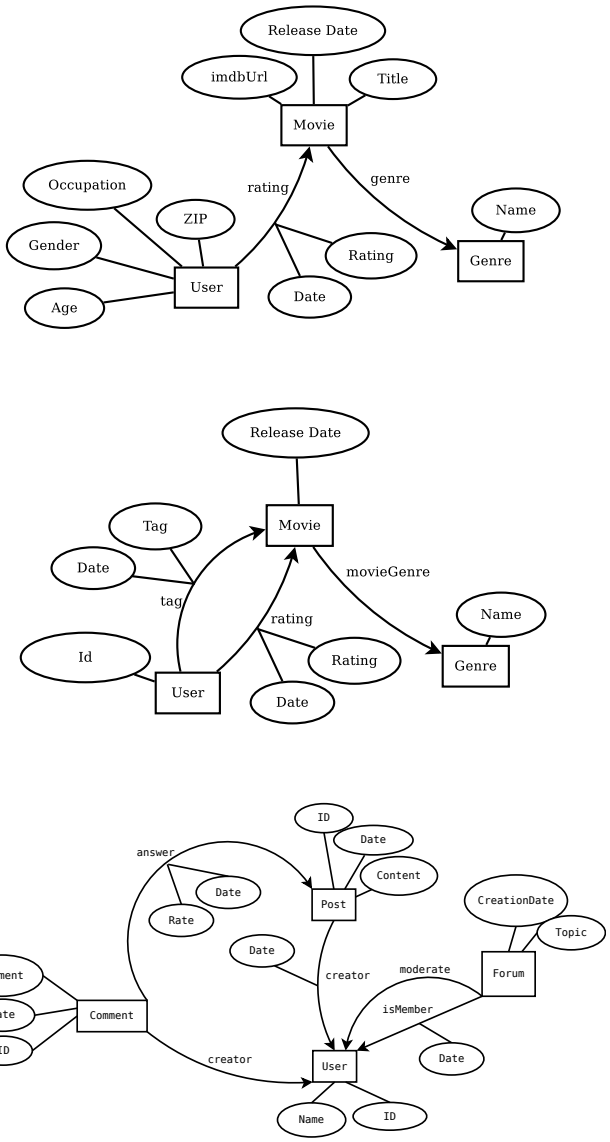

Fig. 7. Attributed graph representing `ML100k` (top), `ML10m` (center), and `SNBsmall` and `SNBlarge` (bottom) datasets.

from 943 users. It also contains 2,874 movie-genre associations for 19 possible genres (Grouplens 2014).

Figure 7 (top) shows the attributed graph representing the small movielens dataset, which we will represent using DEX, Neo4j, HANA Graph and our own structures. The graph model for this dataset has three kinds of entities: users, movies and genres. Movies and users contain attributes presenting different value distributions. Regarding to the representation in our structure, we use a k^2-

tree to represent the dense attributes *Age*, *Gender* or *Occupation*, while the remaining sparse attributes are directly stored through an indexed-plain list.

Movielens 10m (`ML10m`) We analyze the results of a different dataset that also represents movie recommendations from Movielens. The model, which is shown in Figure 7 (center), contains a more reduced set of properties. However, the number of entities is larger than the previous use case. This dataset contains 10,000,053 ratings for 10,681 movies from 71,567 users (Grouplens 2014), where more than 20 million of properties need to be stored. It also contains 21,564 movie-genre associations for 20 possible genres, and 95,581 tags given by users to movies.

LDBC-SNB (`SNBsmall` and `SNBlarge`) We also create two synthetic datasets using the LDBC Social Network Benchmark (Erling et al. 2011) (LDBC-SNB). The model, which is shown in Figure 7 (bottom), simulates a realistic social network containing three type nodes (Users, Posts, Forums, and Comments) and several edge types (answer, creator, isMember, moderator). We tune the configuration files to vary cardinalities and distributions, thus generating two graph datasets of different size, `SNBsmall` and `SNBlarge`. The small variant contains 336,428 nodes (184,091 comments, 16,256 forums, 4,515 people, 131,566 posts) and 848,873 edges from different types. The large variant contains 10,547,201 nodes (7,329,735 comments, 249,935 forums, 27,007 people, and 294,0524 posts) and 26,700,489 edges from different types. Regarding the representation in our structure, we use a k^2-tree to represent the dense attributes *Topic* and *Date* at the nodes, and the attribute *Rate* for edges of type *Answer*, while the remaining attributes are considered sparse and are directly stored through an indexed-plain list.

7.2. Results

We first show the spatial cost for representing all datasets for the five different approaches. Figure 8 shows the cost in megabytes. Note that in the case of AttK^2-tree and dynAttK^2-tree, we show the memory usage in main memory, while in the case of Dex and Neo4j the results are measured as their cost in secondary memory with the graph engine system offline. For HANA Graph, we use the estimated maximum memory consumption data reported in the table run information. Our proposal achieves better spatial results than DEX and Neo4j. Compared to HANA Graph, we obtain slightly larger sizes. However, we will see in the following temporal comparisons that this solution is much worse in terms of time performance.

Figure 9 shows some temporal results of simple and fast operations over the schema of the graph. More concretely, we ask for the type of a node or an edge (queries *GetNodeType* and *GetEdgeType*). They are very lightweight operations in our system, particularly for AttK^2-tree, since each type has a range of identifiers. Thus, given an identifier, we only need to search over the list of node or edge types, which usually contains very few elements. In the case of dynAttK^2-tree, this operation is solved using only an access operation over a dynamic wavelet tree, which is efficient in practice. Only DEX obtains similar results to the dynamic version when solving *GetEdgeType* over the largest dataset (`SNBlarge`). It

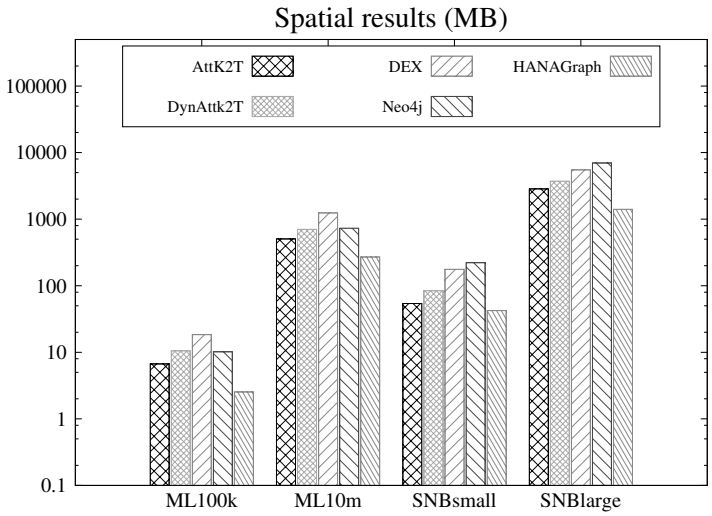

Fig. 8. Spatial results obtained by each system over the four datasets.

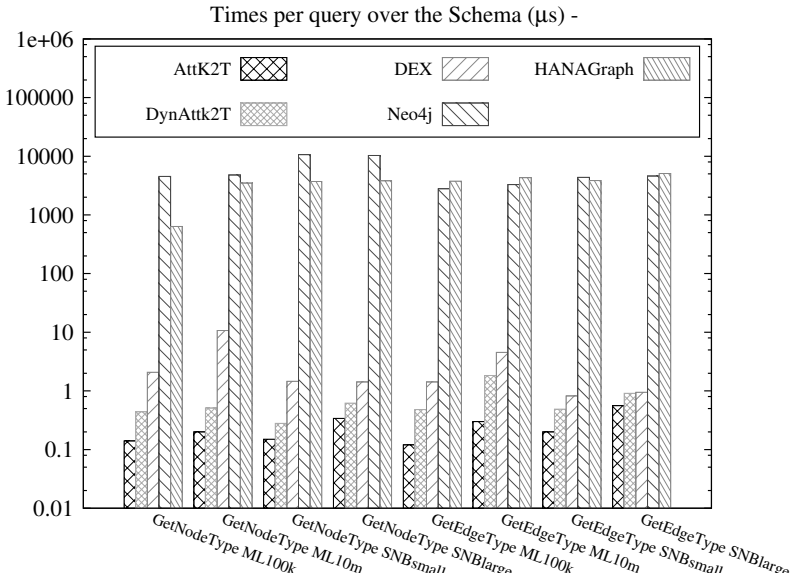

Fig. 9. Time results obtained for operations over the schema: *GetNodeType, GetEdgeType.*

is important to mention that in the case of DEX, Neo4j, or HANA Graph, as we described in Section 7.1.1, every query performed in our experimental evaluation involves the parsing of the query, the connection with the database and other operations.

Figure 10 shows the results obtained for the operations over the graph data.

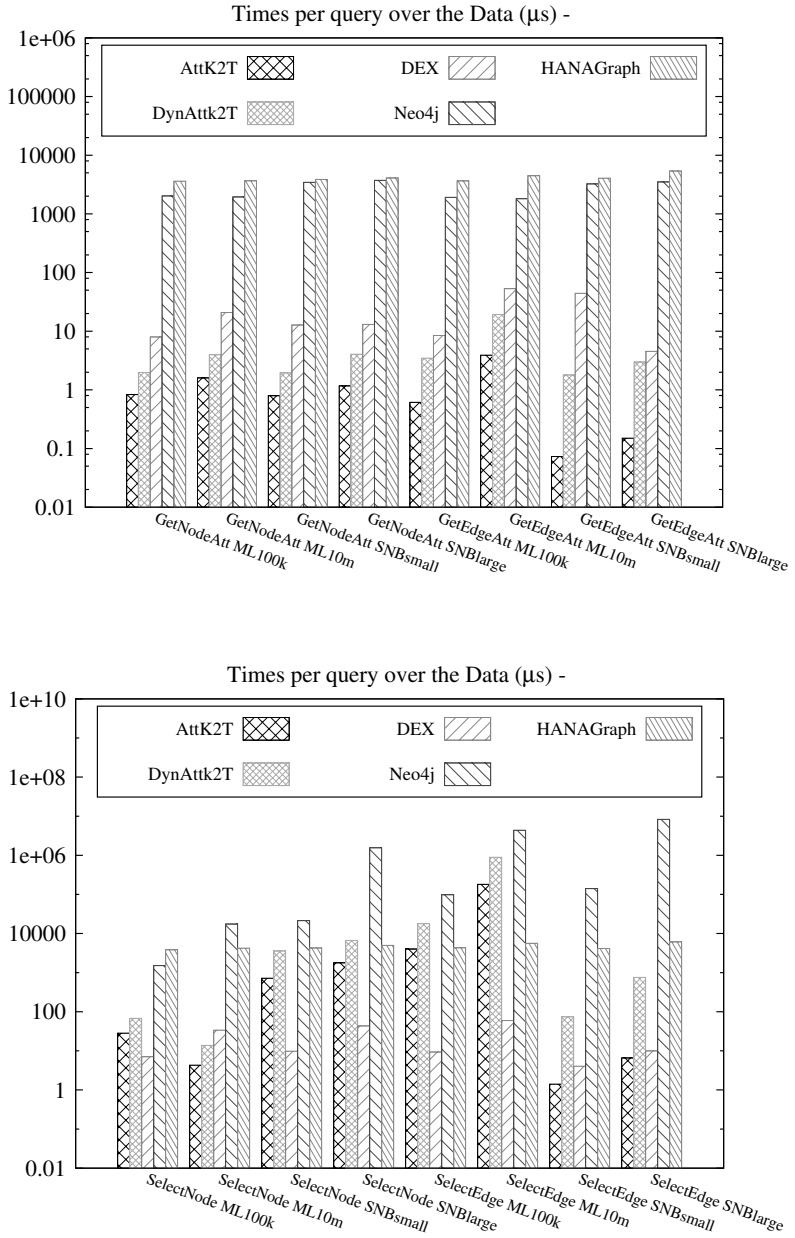

Fig. 10. Time results obtained for operations over the data: *GetNodeAttribute, GetEdgeAttribute*(top) and *SelectNode, SelectEdge* (bottom).

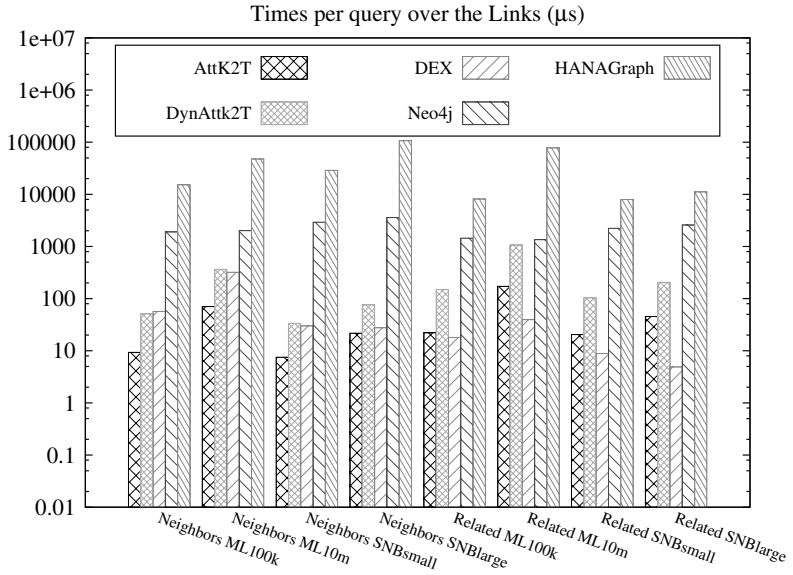

Fig. 11. Time results obtained for operations over the links: *Neighbors, Related*.

In both of our variants, AttK^2-tree and dynAttK^2-tree, we can see how obtaining the property value for a given node/edge (operations *GetNodeAttribute* and *GetEdgeAttribute*) is faster than obtaining the list of nodes/edges that take a given value (operations *SelectNode* and *SelectEdgee*). Compared with the other systems, our proposals are faster for *GetNodeAttribute* and *GetEdgeAttribute*. However, they are outperformed by DEX when solving *SelectNode* and *SelectEdge* over most of the datasets. Neo4j and HANA Graph are slower for all the operations. AttK^2-tree and dynAttK^2-tree obtain better results than DEX for dataset `ML10m`. This is due to the fact that all node attributes are sparse, and they are faster to retrieve than dense attributes for our proposal. This happens also with edge attributes for the synthetic datasets (`SNBsmall` and `SNBlarge`), as most of them are stored as sparse attributes; thus, AttK^2-tree obtains better results than DEX when performing *SelectEdge*.

Finally, Figure 11 shows operations over the relations among nodes. Links in AttK^2-tree and dynAttK^2-tree are stored using a k^2-tree (and an extra structure of bitmaps), so these operations are very fast in our structures, since they do not involve operations over the attributes. Operation *Related* is slower than *Neighbors* for AttK^2-tree and dynAttK^2-tree, because *Related* implies an additional filtering of the final list of candidate edges, which is not necessary in the case of the *Neighbors* operation. DEX system improves the results obtained by our proposal, while Neo4j and HANA Graph are slower in both operations for all datasets.

7.3. Summary of results

AttK^2-tree is a very compressed representation of attributed graphs that supports efficient access to the properties and the relationships of the elements of the graph. The structure we propose is not a full graph database engine, as we only support a set of basic queries. However, the spatial and temporal results obtained in the experimental evaluation show that it is a very competitive approach to represent static graph data in a very compact way and to perform basic graph operations over the compressed structure. The dynamic variant achieves worse space/time results compared to the static alternative, as expected. However, compared to the rest of the graph database systems, which also allow flexible schemas and modifications of the data, it obtains a very good compromise between space requirements and query performance.

8. Conclusions and future work

We presented a new compact representation of attributed graphs that supports efficient access to the nodes, edges and their properties. More concretely, our proposal is designed for representing and navigating labeled, directed, attributed multigraphs in little space and efficient time. It works in main memory and it relies on the k^2-tree structure for representing most of the data. Relations among nodes are represented using an extension of the k^2-tree that supports multiple edges among the same pairs of nodes. Regarding to the properties of the elements, we differentiate among dense attributes (presenting very few different values), which are stored using k^2-trees, and sparse attributes, stored as plain lists.

We presented two different variants of the representation: a static version, denoted AttK^2-tree, and a dynamic version, denoted dynAttK^2-tree. We experimentally evaluated the spatial and temporal performance of both of our variants for datasets of different nature and size. We also stored the same data in DEX, Neo4j, and HANA Graph in order to provide some spatial and temporal references of other systems. Results showed that our proposals obtain competitive space and time results, compared with these existing graph database management systems. However, it is important to note that these systems are full attributed graph engines with many features and possible combinations, in addition to complete query APIs, so this comparison has to be understood only as a proof of concept of our structure.

The k^2-tree is the basis of the proposed approach, as it is used for representing the binary relations among nodes and also among nodes/edges with attribute values. The compact spaces obtained by our solution are due to the good properties of k^2-trees in terms of space. In any case, our solution can be regarded as a modular system, where each part can be replaced with other compact data structures that improve the space/time trade-off. For instance, instead of k^2-trees, one could explore other promising representations that have appeared recently for graph compression (Fischer and Peters 2016, Maneth and Peternek 2016), as soon as they become mature enough to compete with k^2-tree not only in terms of space, but also in terms of scalability and extended functionality.

We implemented a basic set of operations to query the properties and the connections of the elements of the graph. A future line of research will include the design and implementation of algorithms to solve more complex operations. In addition, we will explore the influence of node and edge reordering in our

proposal. As studied by Brisaboa et al. (2014), node ordering is a key aspect for obtaining high compression in the k^2-tree structure. In the case of AttK^2-tree, nodes and edges are sorted according to their labels, in a decision made to save space and improve performance for some queries. This is not further used in the dynamic variant, dynAttK^2-tree, as node and edge identifiers are given sequentially as they are inserted. Thus, using the same data structures as the dynamic variant for storing node/edge types, we can reorder node/edge identifiers in the static version, trying to minimize space consumption.

Acknowledgements

This research has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie [grant agreement No 690941]; from the Ministerio de Economía y Competitividad (PGE and ERDF) [grant numbers TIN2015-69951-R; TIN2016-77158-C4-3-R] and from Xunta de Galicia (co-founded with ERDF) [grant numbers ED431C 2017/58; ED431G/01]. We also thank Nieves R. Brisaboa for her contributions during the initial discussions of this work.

References

Aggarwal, C. and Wang, H. (2010), *Managing and Mining Graph Data*, Springer.

Álvarez-García, S., de Bernardo, G., Brisaboa, N. and Navarro, G. (2017), 'A succinct data structure for self-indexing ternary relations', *Journal of Discrete Algorithms* **43**, 38–53.

Angles, R. and Gutiérrez, C. (2008), 'Survey of graph database models', *ACM Computing Surveys (CSUR)* **40**(1), 1.

Böhm, H.-J. and Schneider, G. (2000), *Virtual Screening for Bioactive Molecules*, Wiley.

Boldi, P. and Vigna, S. (2004), The WebGraph framework I: Compression techniques, *in* 'Procs. of the 13th International World Wide Web Conference (WWW)', pp. 595–601.

Bornea, M., Dolby, J., Kementsietsidis, A., Srinivas, K., Dantressangle, P., Udrea, O. and Bhattacharjee, B. (2013), Building an efficient RDF store over a relational database, *in* 'Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD)', ACM, pp. 121–132.

Brisaboa, N., Cerdeira-Pena, A., de Bernardo, G. and Navarro, G. (2017), 'Compressed representation of dynamic binary relations with applications', *Information Systems* **69**, 106–123.

Brisaboa, N., Ladra, S. and Navarro, G. (2014), 'Compact representation of web graphs with extended functionality', *Information Systems* **39**(1), 152–174.

Caro, D., Rodríguez, M. A., Brisaboa, N. R. and Fariña, A. (2016), 'Compressed kd-tree for temporal graphs', *Knowledge and Information Systems* pp. 553–595.

Chierichetti, F., Kumar, R., Lattanzi, S., Mitzenmacher, M., Panconesi, A. and Raghavan, P. (2009), On compressing social networks, *in* 'Procs. of 15th Conference on Knowledge Discovery and Data Mining (KDD)', pp. 219–228.

Ciglan, M., Averbuch, A. and Hluchy, L. (2012), Benchmarking traversal operations over graph databases, *in* 'Procs. of the 28th International Conference on Data Engineering Workshops (ICDEW)', pp. 186–189.

Claude, F. and Navarro, G. (2010), 'Fast and compact Web graph representations', *ACM Transactions on the Web (TWEB)* **4**(4), article 16.

Conte, D., Foggia, P., Sansone, C. and Vento, M. (2004), 'Thirty years of graph matching in pattern recognition', *Int. Journal of Pattern Recognition and Artificial Intelligence* **18**(3), 265–298.

de Bernardo, G., Álvarez-García, S., Brisaboa, N., Navarro, G. and Pedreira, O. (2013), Compact querieable representations of raster data, *in* 'Proc. 20th International Symposium on String Processing and Information Retrieval (SPIRE)', LNCS 8214, pp. 96–108.

Erling, O., Averbuch, A., Larriba-Pey, J., Chafi, H., Gubichev, A., Prat, A., Pham, M.-D.

and Boncz, P. (2011), The LDBC social network benchmark: Interactive workload, *in* 'Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD)', ACM, pp. 619–630.

Fischer, J. and Peters, D. (2016), 'GLOUDS: Representing tree-like graphs', *Journal of Discrete Algorithms* **36**, 39–49.

Grouplens (2014), 'Movielens dataset'. [Online; accessed 27-August-2018].
URL: *http://grouplens.org/datasets/movielens/*

Gyssens, M., Paredaens, J., Van den Bussche, J. and Van Gucht, D. (1994), 'A graph-oriented object database model', *IEEE Transactions on Knowledge and Data Engineering* **6**(4), 572–586.

Han, J., Haihong, E., Le, G. and Du, J. (2011), Survey on NoSQL database, *in* 'Procs. of the 6th International Conference on Pervasive Computing and Applications (ICPCA)', pp. 363–366.

Hernández, C. and Navarro, G. (2014), 'Compressed representations for web and social graphs', *Knowledge and Information Systems* **40**(2), 279–313.

Iordanov, B. (2010), HyperGraphDB: a generalized graph database, *in* 'Web-Age Information Management', Springer, pp. 25–36.

Jacobson, G. (1989), Space-efficient static trees and graphs, *in* 'Procs. of the 30th IEEE Symposium on Foundations of Computer Science (FOCS)', pp. 549–554.

Ladra, S., Paramá, J. and Silva-Coira, F. (2017), 'Scalable and queryable compressed storage structure for raster data', *Information Systems* **72**, 179–204.

Larriba-Pey, J. L., Martínez-Bazán, N. and Domínguez-Sal, D. (2014), Introduction to graph databases, *in* 'Reasoning Web. Reasoning on the Web in the Big Data Era', Vol. 8714 of *Lecture Notes in Computer Science*, Springer International Publishing, pp. 171–194.

Levene, M. and Poulovassilis, A. (1990), The hypernode model and its associated query language, *in* 'Procs. of the 5th Jerusalem Conference on Information Technology', IEEE, pp. 520–530.

Mäkinen, V. and Navarro, G. (2008), 'Dynamic entropy-compressed sequences and full-text indexes', *ACM Transactions on Algorithms* **4**(3), article 32. 38 pages.

Malewicz, G., Austern, M. H., Bik, A. J. C., Dehnert, J. C., Horn, I., Leiser, N. and Czajkowski, G. (2010), Pregel: A system for large-scale graph processing, *in* 'Procs. of the 2010 ACM International Conference on Management of Data (SIGMOD)', pp. 135–146.

Maneth, S. and Peternek, F. (2016), Compressing graphs by grammars, *in* 'Procs. of the 32nd IEEE International Conference on Data Engineering (ICDE)', IEEE, pp. 109–120.

Martínez-Bazan, N., Águila-Lorente, M. A., Muntés-Mulero, V., Dominguez-Sal, D., Gómez-Villamor, S. and Larriba-Pey, J. L. (2012), Efficient graph management based on bitmap indices, *in* 'Procs. of the 16th International Database Engineering & Applications Sysmposium (IDEAS)', ACM, pp. 110–119.

Martínez-Bazan, N., Muntés-Mulero, V., Gómez-Villamor, S., Nin, J., Sánchez-Martínez, M. A. and Larriba-Pey, J. L. (2007), DEX: High-performance exploration on large graphs for information retrieval, *in* 'Procs. of the 16th ACM conference on Conference on Information and Knowledge Management (CIKM)', ACM, pp. 573–582.

Navarro, G. (2014), 'Wavelet trees for all', *Journal of Discrete Algorithms* **25**, 2–20.

Navarro, G. (2016), *Compact Data Structures – A practical approach*, Cambridge University Press. ISBN 978-1-107-15238-0. 570 pages.

Padrol-Sureda, A., Perarnau-Llobet, G., Pfeifle, J. and Muntés-Mulero, V. (2010), Overlapping community search for social networks, *in* 'Procs. of the IEEE 26th International Conference on Data Engineering (ICDE)', IEEE Press, pp. 992–995.

Paradies, M., Kinder, C., Bross, J., Fischer, T., Kasperovics, R. and Gildhoff, H. (2017), GraphScript: implementing complex graph algorithms in SAP HANA, *in* 'Procs. of the 16th International Symposium on Database Programming Languages (DBPL)', ACM, pp. 13:1–13:4.

Prezza, N. (2017), A framework of dynamic data structures for string processing, *in* 'International Symposium on Experimental Algorithms', Leibniz International Proceedings in Informatics (LIPIcs).

Raghavan, S. and Garcia-Molina, H. (2003), Representing web graphs, *in* 'Procs. of the IEEE 19th International Conference on Data Engineering (ICDE)', IEEE Press, pp. 405–416.

Robinson, I., Webber, J. and Eifrem, E. (2013), *Graph Databases*, O'Reilly.

Samet, H. (2006), *Foundations of Multidimensional and Metric Data Structures*, Morgan Kaufmann Publishers Inc.

SAP (2016), 'SAP HANA Graph Reference'. Document version: 1.0.

Sun, W., Fokoue, A., Srinivas, K., Kementsietsidis, A., Hu, G. and Xie, G. (2015), SQLGraph: An efficient relational-based property graph store, *in* 'Procs. of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD)', ACM, pp. 1887–1901.

Tinkerpop (2014), 'Gremlim query language'. Available: https://github.com/tinkerpop/gremlin/wiki.

Author Biographies

Sandra Álvarez-García is a software analyst and engineer at Indra and a collaborating researcher of the Database Laboratory at the University of A Coruña. She received her degree in Computer Science Engineering in 2009 and her Ph.D. degree in Computer Science in 2014, both from the University of A Coruña. Her research was mainly focused on obtaining compressed and efficient representation of graphs, and more particularly, for managing RDF and linked data. Part of her research was carried out at Yahoo! Labs Barcelona and Yahoo! Labs Santiago de Chile.

Borja Freire obtained his degree in Computer Science at the University of A Coruña in 2016. During his studies, he was awarded with a collaboration grant in the Department of Computer Science. In 2018, he obtained his Master degree in Bioinformatics at the same university. At the same time that he finished his master studies, he was hired by Enxenio S.L. to work in R&D projects related to Bioinformatics. In addition, he was accepted into the Doctorate Program in Computer Science at University of A Coruña, and he has been a Ph.D. student since then.

Susana Ladra is Associate Professor at the University of A Coruña, where she obtained her degree in Computer Science Engineering in 2007 and her Ph.D. degree in Computer Science in 2011 at the same university. She also received her Bachelor in Mathematics from the National Distance Education University (UNED) in 2014. Her fields of interests include the design and analysis of algorithms and data structures, data compression and data mining in the fields of information retrieval and bioinformatics. She has published more than 40 papers in various international journals and conferences and is principal investigator of several national and international research projects.

Óscar Pedreira has M.Sc. and Ph.D. degrees in Computer Science from University of A Coruña. He is an Associate Professor since 2008 at the same institution. He is a researcher of the Database Laboratory. His research interests include algorithms for similarity search, data structures and algorithms for graph databases, geographic information systems, and software engineering. He has co-authored many articles published in journals and conferences relevant for the research areas mentioned. He has continuously participated in research projects and technology and knowledge transfer projects with different companies.

Correspondence and offprint requests to: Susana Ladra, Universidade da Coruña, CITIC, Database Laboratory, Campus de Elviña, 15071, A Coruña, Spain. Email: sladra@udc.es