Cutting-edge Relational Graph Data Management with Edge-k: From One to Multiple Edges in the Same Row

Lucas C. Scabora[1], Paulo H. Oliveira[1], Gabriel Spadon[1], Daniel S. Kaster[2], Jose F. Rodrigues-Jr[1], Agma J. M. Traina[1], Caetano Traina Junior[1]

[1] University of São Paulo (USP), São Carlos, Brazil
{luscascsb,pholiveira,spadon}@usp.br, {junio,agma,caetano}@icmc.usp.br
[2] University of Londrina (UEL), Londrina, Brazil
dskaster@uel.br

Abstract. Relational Database Management Systems (RDBMSs) are widely employed in several applications, including those that deal with data modeled as graphs. Existing solutions store every edge in a distinct row in the edge table, however, for most cases, such modeling does not provide adequate performance. In this work, we propose Edge-k, a technique to group the vertex neighborhood into a reduced number of rows in a table through additional columns that stores up to k edges per row. The technique provides a better table organization and reduces both table size and query processing time. We evaluate Edge-k table management for insert, update, delete and bulkload operations, and compare the query processing performance both with the conventional edge table — adopted by the existing frameworks — and with the Neo4j graph database. Experiments using Single-Source Shortest Path (SSSP) queries reveal that our new proposal approach always outperforms the conventional edge table as well as it was faster than Neo4j for the first iterations, being slightly slower than Neo4j only for iterations after having loaded the whole graph from disk to memory. It was able to reach a speedup of 66% over a representative real dataset, with an average reduction of up to 58% in our tests. The average speedup over synthetic datasets was up to 54%. Edge-k was also the best one when performing graph degree distribution queries. Moreover, the Edge-k table obtained a processing time reduction of 70% for bulkload operations, despite having an overhead of 50% for individual insert, update and delete operations. Finally, Edge-k advances the state of the art for graph data management within relational database systems.

Categories and Subject Descriptors: H.2.4 [**Systems**]: Relational databases; H.2.4 [**Systems**]: Query processing; H.2.1 [**Logical Design**]: Schema and subschema; G.2.2 [**Graph Theory**]: Graph algorithms

Keywords: Graph Management, Query Processing, RDBMS

1. INTRODUCTION

Complex networks are employed to model data in several scenarios, such as communication infrastructures, social networks, and urban street organization [Barabási and Pósfai 2016]. Those networks are commonly represented as graphs, in which nodes are mapped into vertices, and relationships into edges. The number of applications using graph structures to represent data has increased significantly. Many of such applications seek to analyze the characteristics of graphs, employing algorithms such as Page Rank, as well as tackling problems such as finding the Single-Source Shortest Paths (SSSP) and determining the connected components of a graph [Silva et al. 2016].

Assume for example a graph representing authors and their publications, as illustrated in Figure 1. In this hypothetical graph, the relationship between two authors (*e.g.* Researchers A and B) is defined by their common publications, that is, coauthoring the same publication. Suppose that the weight w of the edge corresponding to such relationship is determined as $1/n_p$, where n_p is the number of

This research was supported by FAPESP (Sao Paulo Research Foundation grants #2016/17330-1, #2015/15392-7, #2017/08376-0 and #2016/17078-0), CAPES and CNPq.

publications common to both authors. Then, $0 \leq w \leq 1$, and weights close to 0 represent stronger relationships (stronger lines in Figure 1), whereas weights close to 1 represent weaker relationships (thinner lines). In this scenario, assume that one wants to analyze the collaborative network regarding a specific author seeking to identify their direct and indirect coauthors. This action corresponds to setting the specific author as a starting point (Researcher A in Figure 1) and exploring their direct neighbors (Researcher B), then the next level of neighbors (Researcher C), and so on. The SSSP algorithm can be employed to expand the frontier of visited neighbors from a given starting point.

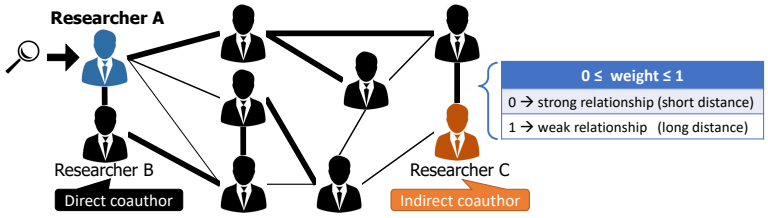


Fig. 1: Example of a collaborative graph representing authors and their publications.

Identifying the proximity between authors and coauthors would allow the analysis of a researcher's impact on their research field. The analysis can consider the weights of the edges explored by the SSSP algorithm. In this case, a weight close to 0 means a short distance between authors, due to a high number of shared publications. On the other hand, a weight close to 1 means a long distance between authors, due to a small number of shared publications. Hence, by identifying the shortest paths from one author to all the others in the graph, the SSSP algorithm is able to determine the strongest connections in the author's collaborative network.

The rapid growth of Web-based applications has led to an increasing volume of graph data [Corbellini et al. 2017]. Such volume is large enough to not fit entirely into main memory, compelling application developers to rely on disk-based approaches. Relational Database Management Systems (RDBMS) provide infrastructure to support graph data management, with some useful features such as storage, data buffering, indexing, and query optimizations. However, they demand additional support to handle graphs. There are several frameworks proposed to manage graphs on top of RDBMSs. The FEM (Frontier, Expand and Merge) framework [Gao et al. 2011; Gao et al. 2014], for instance, was proposed to translate analytical graph queries into Structured Query Language (SQL) commands. The implementation of FEM for SSSP queries consists of tracking frontier vertices and iteratively expanding them, merging the new elements with the set of previously visited ones.

The Grail framework [Fan et al. 2015], as well as its adaptation to the RDBMS Vertica [Jindal et al. 2015], follows a similar approach, detailing query execution plans and allowing query optimizations. However, both frameworks are limited to a single graph modeling, and does not take into account the query execution performance in alternative modelings. Both FEM and Grail frameworks employ one table for the set of edges and another for the set of vertices. The edges are organized as a list, where each edge is stored as one row in the edge table. This representation determines the number of rows required to store a vertex as one row for the vertex itself plus one row per edge, totaling the vertex degree (*i.e.* number of neighbors) + 1.

In this paper, we propose Edge-k, a novel approach to model graphs in an RDBMS. The idea is to store the adjacencies of each vertex using an adjacency list that spans to multiple rows, with a predefined number k of edges per row. Our objective is to avoid tables with either too many rows or too many columns. A reduction in the number of rows decreases the required number of data blocks, which, in turn, reduces the space dedicated to row headers and the number of I/O operations. A reduction in the number of columns contrasts with storing the entire adjacency list in a single row where the number of columns must be equal to the degree of the vertex with the maximum degree in the graph, which may not be known beforehand and may be unnecessarily large for other vertices.

Moreover, by storing adjacencies in the same row, Edge-k ensures that at least part of them will be contiguously stored in the disk, which can further contribute to improve query performance.

Edge-k was initially proposed in [Scabora et al. 2017], where Edge-k was evaluated executing the SSSP procedure. The experiments were evaluated over both real and synthetic datasets, in order to analyze the performance of SSSP query in an open-source RDBMS enhanced with Edge-k. The real dataset correspond to the author collaborations obtained from the *Digital Bibliography & Library Project*[1] (DBLP), and the SSSP procedure ran for 2 to 5 iterations. Considering the synthetic datasets, on the other hand, SSSP ran until its completion, which required 4 iterations. We focused on the SSSP problem due to its major importance for numerous applications, such as to discover indirect relationships in a social network and to find shortest paths to interesting tourist attractions. The results revealed a strong correlation between the average query execution time and the number of data blocks used to store the graph. Thus, reducing the number of blocks, which is related to reducing the number of rows and *null* values, leads to higher performance in queries.

In this paper, we extend our previous work exploring the Edge-k behavior when performing three additional analyses, described as follows.

- ◇ **Insert, update, and delete operations.** In addition to the data retrieval performance, we evaluate the performance of Edge-k in data maintenance operations. Although our approach brings a minor overhead compared to the competitors, these operations tend to be less frequent than query executions. Therefore, the overall impact of the Edge-k table tends to be minimal;
- ◇ **Bulkload operations.** We developed a new technique, executed in the three steps Sort, Cast, and Load, to perform the bulkload operation in Edge-k, where we achieved 70% of performance gain when compared to the bulkload operation over an ordinary edge table;
- ◇ **A more extensive evaluation of data retrieval.** Along with the SSSP queries, we evaluate the performance of Edge-k to retrieve the graph degree distribution. We also compared Edge-k both with the approach commonly employed in RDBMS and with the NoSQL graph database Neo4j. The results show the higher performance of our proposal against the evaluated competitors, allowing a reduction of up to 66% of query processing time at the second iteration of the SSSP procedure, compared to the ordinary edge table over the DBLP dataset, and 58% in the first four iterations.

The remainder of this paper is organized as follows. Section 2 describes the background and related work, covering the existing approaches for graph data management employing RDBMS. Section 3 details the proposed Edge-k storage approach, including its extensions. In Section 4, we present the experimental analyses and discuss the results. Finally, Section 5 concludes the work.

2. BACKGROUND AND RELATED WORK

Graph Definitions. A graph is represented as a pair $G = \{V, E\}$, where V is a set of vertices and E is a set of edges connecting vertices. A path $p = v_0 \overset{p}{\rightsquigarrow} v_n$ in a graph is a sequence of non-repeated edges, in which there is one *source* vertex v_0 and one *target* vertex v_n. The weight of a path is determined by the sum of the weights of its constituent edges. The Single-Source Shortest Path (SSSP) problem consists of finding the shortest path from a source vertex v_0 to each vertex $v_i \in V$, that is, all paths $v_0 \overset{p}{\rightsquigarrow} v_i$ of minimum length. An algorithm that solves the SSSP problem is the Bellman-Ford algorithm [Cormen et al. 2009], used in both FEM and Grail frameworks.

Graph Storage in RDBMSs. Our proposal is based on using the Relational Model to store and retrieve graphs. In this context, edges and vertex are stored as two distinct tables in an RDBMS, one for the vertices and other for the edges. Each vertex in V has a unique identifier attribute (id)

[1] `http://dblp.uni-trier.de/`

and may have as many attributes as needed for its properties. Considering the graph composed of authors and their articles, the properties are the details about authors and their articles. An edge in E is defined as an ordered pair $\langle s, t \rangle$, where s is the *source* vertex *id* and t is the *target* vertex *id*. Additional properties from each edge are stored in extra attributes of the relation, allowing detailing the relationships among the vertices. A common property is the edge weight w_i of each relationship between s and t vertices. In this case, an edge will be defined as $\langle s, t, w_1, \ldots, w_j \rangle$, where j is the number of properties of each edge. Figure 2 illustrates the vertex and edge tables that store the graph presented in Figure 1, where each vertex has only one property: its name; and each edge has only one property: the number of articles that both authors produced together.

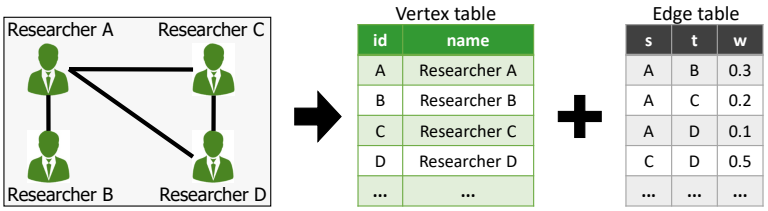


Fig. 2: Example of Graph Storage in RDBMSs considering the author-relationship graph example in Figure 1.

There are three main approaches to store vertex or edges properties using RDBMS [Levandoski and Mokbel 2009; Albahli and Melton 2016]: (i) triple store schema; (ii) binary tables; and, (iii) n-ary tables. The triple store schema organizes vertices and edges into a triple <*subject, property, object*>, in which *subject* refers to the identifier of a vertex or an edge (*e.g.* the respective *id* attribute), *property* defines the attribute of the subject, and *object* is the property value. Following the entity-relationship model, every attribute of both vertices and edges are represented as *properties* in the triple store schema [Neumann and Weikum 2010]. Its major drawback derives from the *object* fields originated from distinct properties, since they may have different data types. The binary tables storage schema groups each distinct *property* into a specific table. As a result, each table has only two columns (hence it is named binary table): the entity identifier and its respective property value [Abadi et al. 2007]. The third schema, *i.e.* the n-ary tables, includes the several properties as additional columns in the same table. A special case of the n-ary table assigns all properties to the same table, which results in a property table [Levandoski and Mokbel 2009].

The main problem of the property table approach (*i.e.*, the complete n-ary table) is the data sparsity caused by heterogeneity in the entity set. DB2RDF [Bornea et al. 2013] was proposed to overcome this drawback, aggregating sets of properties into a single column. DB2RDF defines a maximum number of properties for each row in the vertex table, where each property is represented by a pair of columns denoting its type and the respective value. A hash function will determine which property will be assigned to which column. Moreover, if a property needs to be stored in a column already filled, a new row is inserted in the vertex table to accommodate this new property. The focus of DB2RDF is to reduce the space required to store the dataset since it stores distinct properties in a reduced number of columns. However, DB2RDF focus on only associating vertices with their properties, not dealing with the relationships among vertices and, consequently, with the shortest paths connecting them.

Graph Storage in Non-relational DBMSs. Due to storage problems in Web-based applications, many companies investigate non-relational forms of storing data, resulting in NoSQL alternatives [Corbellini et al. 2017]. NoSQL databases usually hold semi-structured and/or unstructured data, mainly focusing on distributing the data storage and aiming at reducing complex join operations. Graph-oriented NoSQL databases store data in a graph-like structure, in which data are represented by vertices and edges. An example of such databases is the Neo4j[2], an open source project based

[2] `https://neo4j.com/`

on Lucene indexes to store and access data from vertices and their relationships. The graph data stored in Neo4j are accessed through the Cypher or Gremlin languages, or are directly manipulated via a Java API. Studies revealed that Neo4j is more efficient than a relational database (MySQL) when dealing with traversal queries, *i.e.* queries looking for shortest paths with length bigger than 3 [Vukotic et al. 2014; Vicknair et al. 2010]. However, it is important to mention that the performance of Neo4j is negatively impacted by the size of the query result, which can achieve huge sizes for larger graphs and lengths [Vukotic et al. 2014].

An extension of DB2RDF is the SQLGraph [Sun et al. 2015], which employs a similar approach on non-relational databases. SQLGraph focuses on storing the properties of an edge as a JSON object, overcoming the limitation of a fixed maximum number of properties in a single row. The graph transversals and update operations are performed by mixing SQL and Gremlin programming languages. Since SQLGraph does not deal with relational databases (like DB2RDF), the authors compared it with Neo4j and Titan NoSQL databases.

A graph representation that does not comply with the first normal form of the Relational Model stores the complete adjacency lists of each vertex in a column of type array [Chen 2013]. In this approach, each row has the *source* vertex *id* and its respective neighborhood (*i.e.* its list of *target* vertex *id*) in a single table column. According to the authors, this representation is well suited for sparse graphs since it avoids *null* values. However, such proposal only deals with scenarios in which all vertices have a small degree. As a consequence, the authors did not explore scenarios in which the neighborhood column size is bigger than a data block size.

Considerations. Our proposed Edge-k table focuses on improving RDBMS-based graph processing systems. In other words, instead of employing an array type of any size, we group the neighborhood of a vertex into the edge table according to a maximum of k distinct values. Since our proposal covers a specialization of an edge table, it differs from both DB2RDF and SQLGraph because they either focus on the vertex table or manage multiple edge tables in the same structure. Thus, our approach gathers the main advantages of the existing approaches while it strictly follows the Relational Model.

3. THE EDGE-K TABLE ORGANIZATION

There are two main approaches to store the edges of a graph as an RDBMS table. The first is to store one edge per row; the problem with this approach is the need to scan tables with too many tuples. The second approach is to store the complete adjacency list of each node as a row; the problem here is that, each vertex has a different number of neighbors, thus the number of columns must be the largest vertex degree of the graph, which often leads to a table with a very high number of *null* values (as usually few nodes have such number of vertex). In both cases, the access and processing times per row are expensive; our proposal solves both problems, described as follows.

Given a directed graph, our method stores the vertices in a table, and the edges in an Edge-k table. The tuples of the Edge-k table represent the edges as an adjacency list, but limiting the number of edges to a maximum of k entries per tuple. When a *source* vertex has more than k edges, the adjacency list is divided into several tuples of at most k edges, as shown in Figure 3. The columns of the Edge-k table represent the *source* vertex, each of its *target* vertices, and the properties (or weights) of each corresponding edge. Notice that there will be k columns for each additional edge property, and distinct properties will not be stored in the same column. Formally, assuming that only one property is stored per edge, an Edge-k table is represented as $E_k = \{s, \langle t_1, w_1\rangle, \langle t_2, w_2\rangle, \langle t_3, w_3\rangle, ..., \langle t_k, w_k\rangle\}$, where w_i is the property weight of an edge linking vertex s to vertex t_i, and $1 \leq i \leq k$.

Since our proposal stores subsets of the vertex neighborhood in the same row, it enables the reduction of the storage space, the number of disk accesses and the query response time, avoiding the aforementioned problems. Figure 3 illustrates part of the Edge-k table for a graph with a *source*

vertex v_0 and a neighborhood of size $n > k$. Each edge has a property w. The left table shows the conventional approach in RDBMSs, which stores only one edge per row. The right table is a Edge-k table, which stores up to k edges originated from a single vertex at the same row. Notice that this example is defined over a directed graph. However, our proposal also works on undirected graphs by replicating the edges, and inverting the *target* and the *source* vertex *id*.

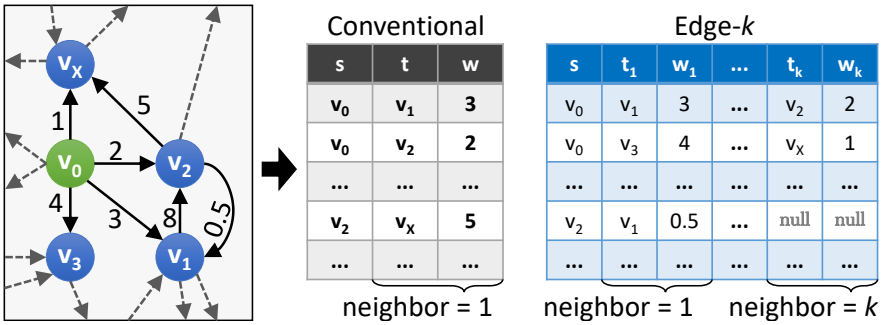


Fig. 3: Comparison between the conventional and proposed approaches.

Our method differs from former alternatives by using the number k of columns as a parameter, which fosters reducing the number of *null* values. The efficiency of our proposal is related to the parameter k, which affects (i) the number of rows and (ii) the unoccupied space of the table (*null* values). Reducing both (i) and (ii) decreases the number of tuples to be stored, consequently improving the query performance. Since there can be mismatches between the neighborhood size n and the chosen k value, it is necessary to define rules to manipulate edges in the Edge-k table, discussed as follows:

- ◇ **n = k:** The ideal case is when n and k are equal, where a single completely-occupied row contains the whole vertex neighborhood (without any *null* values).
- ◇ **n < k:** When n is smaller than k, all the corresponding neighborhood fits in a single row and the remaining values are filled with *null* values. In Figure 3 this can be observed for *source* vertex v_2, which requires *null* values to complete its row.
- ◇ **n > k:** For values of n larger than k, the neighborhood is stored in groups of size k. The last group stores at most k neighbors (second case) and thus the row is completed with *null* values. This last case is illustrated in Figure 3 by the *source* vertex v_0, which have neighborhood size $n = 2k$. In this example, the neighborhood is divided into two complete rows (first case).

The details of our proposal are described as follows. Section 3.1 describes the equations used to measure the impact of the parameter k. Section 3.2 details how to manage an Edge-k table considering insert, update, delete and bulkload operations. Finally, Section 3.3 details how to adapt SSSP queries to execute over an Edge-k table.

3.1 Estimation of the number of rows and of *null* values

In this section, we evaluate how to estimate the number of rows and *null* values that arise when using the Edge-k technique. The number of rows occupied by each *source* vertex v is defined as the smallest integer greater than its total neighborhood size ($degree_v$) divided by k. Therefore, the total number of rows η_{rows} in the table is obtained as the summation of the number of rows for each $v \in V$, as expressed in Equation 1.

$$\eta_{rows} = \sum_{v \in V} \left\lceil \frac{degree_v}{k} \right\rceil \tag{1}$$

The number of *null* values for each *source* vertex v is the number of remaining cells that were not filled with *target* vertices data. This number is obtained computing how many *target* vertices are in that row, which is the rest of the integer division of $degree_v$ by k. The number of columns filled with *null* values for each vertex v is obtained by subtracting the number of *target* vertices previously calculated from k. A special case of this equation is when $degree_v = k$, so the number of columns filled with *null* values is the rest of its division by k. The total number of *null* values is obtained by the summation of the number of *nulls* for each $v \in V$, as is described in Equation 2.

$$\eta_{nulls} = \sum_{v \in V} (k - (degree_v \bmod k)) \bmod k \tag{2}$$

The optimal scenario for Edge-k table is having every row fully filled for every *source* vertex. In this case, both the row overhead imposed by the row header (necessary to store internal control data, *e.g.* the 23 bytes of alignment padding) and the space lost by *null* values are minimum. However, such a scenario could only be achieved for the particular case in which every vertex has a fixed number of neighbors. Unfortunately, this is not true for most graphs. For that reason, we use the metric *exceeding* to evaluate the impact of a given choice of k; it embodies the number of *null* values, redundant index data, and header data into a single number that indicates the cost of a k configuration. Equation 3 computes the *exceeding* metric. Notice that the terms η_{nulls} and η_{rows} are both function of k.

$$exceeding = (\eta_{nulls} * null\_size) + (\eta_{rows} - |V_{neigh}|) * (size(v_{id}) + overhead) \tag{3}$$

where $V_{neigh} \subseteq V$ is the subset of vertices containing a neighborhood; $size(v_{id})$, *overhead* and $null\_size$ are the sizes in bytes respectively of the vertex *id*, of the row overhead and of the storage of a *null target*. The first part of Equation 3 computes the space occupied by *null* values given by the product of η_{nulls} and of the space occupied by each *null* value ($null\_size$). The second part considers the wasted space occupied by replicating *source* vertices v_{id}, which occurs whenever the neighborhood size is greater than the chosen k value. In other words, $\eta_{rows} - |V_{neigh}|$ corresponds to the number of extra rows in an Edge-k table, which is multiplied by the cost of each additional row. The cost of every additional row is defined by the replicated *source* vertex v_{id} plus the row overhead. Finally, Equation 3 sums these two parts to determine the total *exceeding* space.

The value of $null\_size$ depends on how the RDBMS implements *null* values. When it does not compact *null* values, it is the standard size in bytes of the corresponding attribute. In this case, $null\_size = size(v_{id})$. However, an RDBMS usually applies techniques to save space when storing *null* values, and the real size varies according to the product. Although Equations 1 to 3 help users to tune the storage of graphs for every scenario, our proposal depends on choosing an informed value for k to allow better results. In our experiments, we explored various values of k, analyzing their impact on the Edge-k table and on the query processing time (see Section 4.2).

3.2 Edge-k Table Management

This section details how to manage an Edge-k table considering the operations of insert, update and delete edges. Executing those operations over an Edge-k table is more expensive than over a conventional table because they demand not only locating a row, but also finding its respective column (*target* vertex) in that row. Each operation is implemented as a procedure, detailed as follows.

To execute an edge insert operation, it is required to search for an empty spot in the rows of Edge-k table already storing edges from the same *source* vertex. If an available spot exists, the corresponding *target* vertex will be written to the last row of the *source* vertex along with its properties, creating a new *target* vertex, and its associated properties. Otherwise, a new row will be created, inserting the

new edge and filling up the remaining spots with *null* values. For the update operation, the approach is similar, however searching for the column pointing to a specific *target* vertex and updating its property values. Here, we assume that the update operation refers solely to changing some property of a given edge, such as the weight value. Updating the *source* or *target* vertex is equivalent to a deletion followed by the creation of a new edge.

The delete operation is more complex. It always tries to maximize the number of complete rows, that is, maintain rows without any *null* values. For example, consider the deletion of an edge between *source* vertex v_s and *target* vertex v_t. This edge will be deleted by assigning *null* to the column corresponding to v_t in a row r. There are two cases: (i) Either row r is incomplete or all rows of vertex v_s are complete. In this case, we just swap the value in the column of vertex v_t with the value from the last occupied column in the same row; (ii) There is an incomplete row r'. In this case, the swap operation occurs between r and r', replacing the last occupied column in row r' with the value from the column corresponding to v_t in row r. Afterward, the columns corresponding to v_t and their respective properties are filled with *null* values. If such operation removes the last *target* vertex in the row, this row is removed from the table.

We also developed a bulkload operation for the Edge-k table. This operation receives as input a comma-separated file (*i.e.* a CSV file, containing each edge in a distinct row, as shown in Figure 2) and is executed in three steps: **Sort:** a disk-based sort operation over the lines of the CSV file based on the *source* vertex *id*; **Cast:** convert the sorted file, with one edge per line, into another CSV file containing k edges per line; and **Load:** execute the bulkload in the RDBMS using the converted file. The sorting step is required because, in the second step, we sequentially read the CSV file lines and combine them in the same row as long as the *source* vertex remains the same.

3.3 Adaptation of Single-Source Shortest Path (SSSP) procedure

The SSSP query is a search-find greedy-based algorithm. It starts at a user-given location (*i.e.* the *source* vertex), from which it will iteratively expand the vertex's neighborhood (or frontier) until it reaches a given stop criterion. The SSSP algorithm can run: (a) until it calculates the distances to all the vertices of the network; (b) until it finds the desired destination; or (c) until it completes a certain number of iterations. The three cases are achieved through the process of vertex expansion, which implies in searching the neighborhood of another vertex that pertains to the current vertex, such that, at each iteration, the neighborhood of a new vertex will be searched, which becomes the new current *source* vertex. To perform the SSSP algorithm on a Edge-k table, we adapted the procedure commonly employed in the literature. Algorithm 1 describes the SSSP query using our proposal, and Figure 4 illustrates its execution.

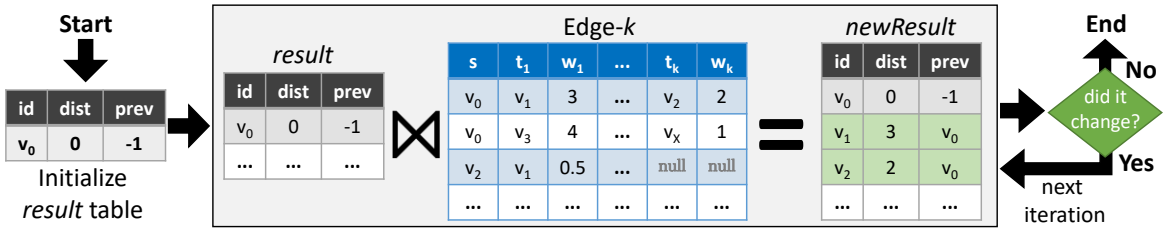


Fig. 4: Illustration of the SSSP procedure adapted to our proposal.

Initially the SSSP procedure creates a temporary table, called `result`, intended to register all the visited vertices. The `result` table keeps track of the unique identifier (*id*) of each vertex, its minimum distance to the source vertex (*dist*), and the previous vertex (*prev*) in the respective shortest path, allowing the identification of the edges that compose such path. This initial step inserts the *source* vertex v_0 into `result`, with a distance of 0 and no previous vertex (*i.e.* -1). This step is represented

in the leftmost part of Figure 4. Step 1 corresponds to Lines 2–4 of Algorithm 1, which initializes the *continue* flag, the *currentIteration* counter, and the `result` table.

Thereafter, the procedure performs a *while* loop (Lines 5–11 of Algorithm 1), depicted in Figure 4 as the operations within the gray box and the condition evaluation at the right. In Line 5, the loop checks whether the *continue* flag is active, as well as compares the *currentIteration* counter to the maximum number of iterations provided by the user. If both conditions are met, `result` is joined with the Edge-k table, expanding the frontier of visited vertices and updating the distance of the shortest path to each *target* vertex. This step produces a new temporary table, called `newResult` (Line 6). To update the distance of a path, our procedure sums the accumulated distance (*dist* in `result`) and the distance w_i (stored in the Edge-k table) to the next neighboring vertex t_i, where $1 \leq i \leq k$. Regarding the example in Figure 4 (corresponding to the graph of Figure 3), after the first iteration, the weights of paths $v_0 \overset{p}{\rightsquigarrow} v_1$ and $v_0 \overset{p}{\rightsquigarrow} v_2$ are, respectively, $(0 + 3)$ and $(0 + 2)$. Our procedure seeks to minimize such weights, keeping the minimum paths found so far. Therefore, in the second iteration, when the procedure visits vertex v_2, the weight of path $v_0 \overset{p}{\rightsquigarrow} v_1$ is updated to 2.5.

Then, in Line 7, the procedure checks whether the contents of `result` and `newResult` tables are the same, counting the number of distinct rows and storing it in the *continue* flag. That is, if the contents are equal, then *continue* equals 0. In this case, when the procedure checks the loop conditions again (Line 5), the loop will be over, finishing the SSSP execution. After updating the *continue* flag, the procedure drops the `result` table, makes `newResult` the up-to-date `result` table, and increments *currentIteration* (Lines 8–10). Then, another iteration begins.

Algorithm 1 Execution of SSSP in Edge-k

```
1: function SSSP(Edge-k, sourceVertex, maxIterations)
2:     continue ← 1
3:     currentIteration ← 0
4:     Create and Initialize result table with sourceVertex
5:     while (continue > 0) AND (currentIteration < maxIterations) do
6:         Visit neighbors of current node by joining tables result and Edge-k, resulting in a new table named newResult
7:         Do a NATURAL FULL JOIN between result and newResult to check whether their contents are the same, counting
       the number of distinct rows and storing it in continue
8:         Drop table result
9:         Rename newResult to result
10:        currentIteration ← currentIteration + 1
11:    end while
12: end function
```

The main difference of Edge-k compared to other proposals is the join of the temporary `result` table with the edge table. In the existing frameworks, the edge table stores only one *target* vertex per row. Therefore, when joining the `result` and edge tables, the other procedures just performs an ordinary `JOIN` operation. Conversely, in Edge-k, for any value of k, such `JOIN` operation features an unnesting step, which consists of mapping the multiple $t_1 \dots t_k$ columns from the Edge-k table into separate rows. Accordingly, Edge-k allows executing `JOIN` operations for any value of k.

4. EXPERIMENTAL RESULTS

4.1 Setup

We evaluated our proposal on both real and synthetic datasets, and we present here the results on representative datasets. The real dataset was extracted from the *Digital Bibliography & Library Project*[3] (DBLP), containing information about authors and their publications. We modeled the set

[3]Dataset from March 16, 2017 — available at `http://dblp.uni-trier.de/xml/`

of authors as vertices and their relationships as edges, measuring the collaboration among them with a weight value (as described in Figure 1). To analyze the impact of varying the number of vertices in the graph and the overall rate of neighboring vertices, we used two synthetic graphs generated with the Networkx[4] framework. The first one, Newman-Watts-Strogatz dataset, shapes a small-world graph. Initially, it creates one ring including all the $|V|$ vertices of the graph. Then, each vertex in the ring may be randomly connected to other X vertices, which creates shortcuts between vertices in the ring structure. The second synthetic graph, Erdos-Renyi dataset, is defined by assigning edges between pairs of vertices with a probability p. Parameters $|V|$, X, and p are provided by the user.

Table I shows the characteristics of the datasets, including the number of vertices and edges, and the minimum, maximum and average degrees of the resulting graphs. For the Newman-Watts-Strogatz graphs, we limited each node to be connected to $X = 200$ nearest neighbors in the ring topology, while varying the number of vertices $|V|$; which revealed to have no effect on the average degree of the network. For the Erdos-Renyi graphs, we modified the probability of edge creation using p equal to 1%, 3%, and 5%, while maintaining the number of vertices static, what lead to a large variation of the vertex degrees. Both scenarios are important because they correspond to multiple cases in which the proposed methods can be validated, which supports a stronger generalization of the results. The experiments were carried out using the RDBMS PostgreSQL 9.5.6 running in a computer equipped with an Intel® Core™ i7-2600 @ 3.40GHz processor, 8GB of DDR3 1,333MHz RAM memory, two SATA 6Gb/s 7,200RPM hard disks set up in RAID 0, and Linux operating system Fedora release 25.

Table I: Characteristics and parameters of the real and synthetic datasets evaluated.

	DBLP	**Newman-Watts-Strogatz**			**Erdos-Renyi**		
Measure	(real)	$X = 200$	$X = 200$	$X = 200$	$p = 1\%$	$p = 3\%$	$p = 5\%$
$\|V\|$	1,909,226	9,000	18,000	27,000	9,000	9,000	9,000
$\|E\|$	19,194,624	1,979,410	3,959,970	5,940,340	810,570	2,428,600	4,048,600
minimum degree	1	203	204	205	56	214	369
average degree	10.05	220.12	219.99	220.01	90.06	269.84	449.84
maximum degree	2100	245	238	247	126	334	536

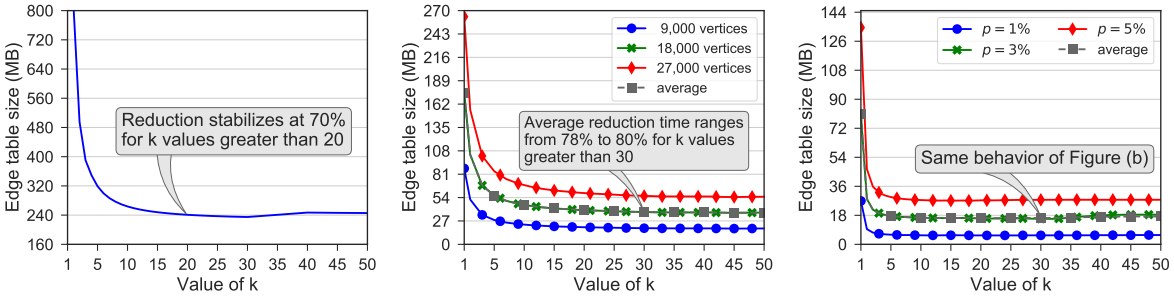
We also provided a comparison between the NoSQL database Neo4j with the PostgreSQL using a conventional edge table and the Edge-k proposed table. For Neo4j, we left the Java Heap Size to be dynamically calculated based on available system resources. We evaluated several experimental setups and assigned the available memory to deliver the best performance for Neo4j in the tests, that is, 4GB for the operating system to reduce disk swap, 2GB for the JVM heap to have enough memory for query execution, and the remaining 2GB to the page cache size.

4.2 Reduction of Dataset Size and SSSP Query Processing Time

This section reviews the analysis of the dataset size and the SSSP query processing time achieved when employing the Edge-k table for varying values of k. Figure 5 shows the resulting dataset size for each dataset. The reduction was about 70% for the DBLP dataset (Figure 5a, for $k \geq 20$), and it ranged from 78% to 80% for both synthetic datasets (Figures 5b and 5c, for $k \geq 30$). It is worth noticing the potential of our approach in terms of table size reduction. As explained by Equation 1, the sharp reduction in the dataset size was analogous to the decrease of η_{rows}, especially when k varies from 1 to 15. For greater values of k, the neighbors of a vertex occupy fewer rows — until most of them eventually fit in one row. In this scenario, as k grows, the number of *null* values also increases, since the Edge-k table becomes larger than required. As *null* values occupy an irrelevant space in the RDBMS PostgreSQL, having more *null* values almost does not impact the total storage size, which is why the dataset size reduction remains stable even for greater values of k.

[4] `https://github.com/networkx/networkx`

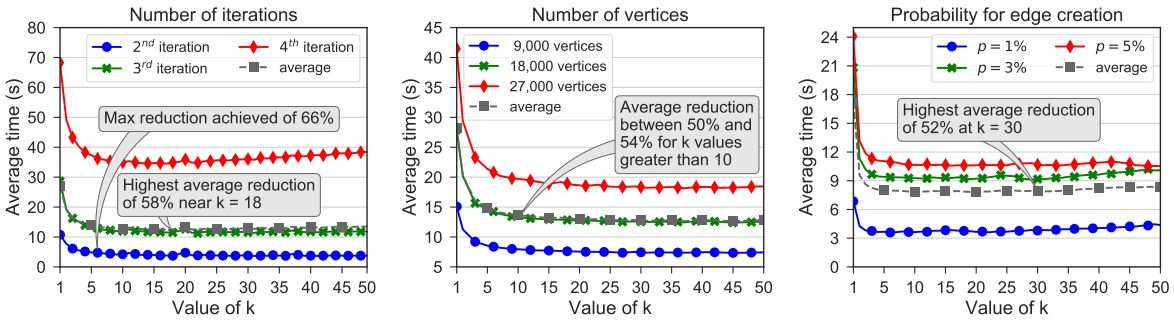
Regarding the SSSP query processing times over the Edge-k table, we defined a query whose starting point was the vertex having the highest degree — the largest number of neighbors. Each query was performed 30 times, and the average execution time was obtained by discarding 10% of the shortest and 10% of the largest results. For the DBLP dataset, we analyzed the average execution times from 2 to 5 iterations of the SSSP algorithm — *i.e.* the number of times the tables Edge-k and `result` are joined to compute shortest paths, as described in Section 3.3. For the synthetic datasets, a maximum of 4 iterations was enough to visit all the vertices. Therefore, we executed the complete SSSP query without limiting the number of iterations.



(a) DBLP. (b) Newman-Watts-Strogatz. (c) Erdos-Renyi.

Fig. 5: Size reduction, in megabytes, of the Edge-k table achieved in each evaluated dataset as k varies.

Figure 6 presents the average query-time to execute a SSSP query varying k when considering the second, third and fourth iterations for the DBLP graph (Figure 6a), for the Newman-Watts-Strogatz graph (Figure 6b) and for the Erdos-Renyi graph (Figure 6c). The SSSP query processing time was not evaluated for the first iteration alone, since the same outcome could be obtained by simply querying the edge table, filtering it by the *source* vertex, rather than performing join operations. The query processing time observed in Figure 6a achieved a maximum reduction of 66% at the 2^{nd} iteration and, regarding the average values, a reduction of 58% when $k \approxeq 18$, while in Figure 6b this reduction varies from 50% to 54% when $k = 10$ and, in Figure 6c, the highest average reduction was about 52% at $k = 30$. Notice that the behavior presented over all datasets is analogous to the dataset size reduction.



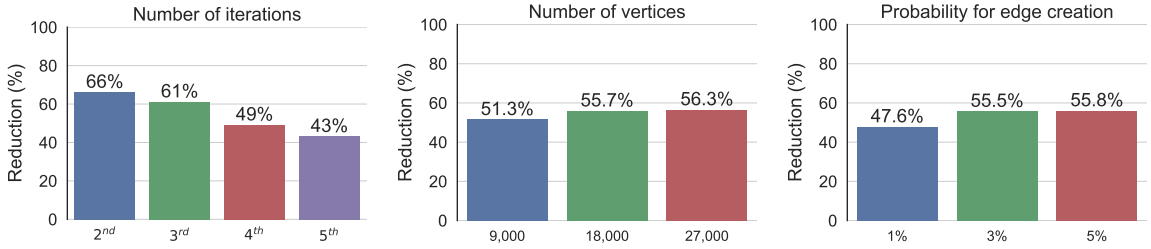
(a) DBLP. (b) Newman-Watts-Strogatz. (c) Erdos-Renyi.

Fig. 6: Average time to process a SSSP query in three scenarios for varying k in the Edge-k table: the DBLP dataset with the second, third and fourth iterations, the Newman-Watts-Strogatz graph and the Erdos-Renyi graph.

Additional experiments, not presented in this paper, were performed for larger values of k. We found that beyond this point the average query time only increases. This is because the number of *null* values rises steadily. In other words, indefinitely increasing the k value also increases the number of unnecessary columns in the Edge-k table, adding more validations to the unnesting operation (see

Algorithm 1). Thus, assigning a value too large for k negatively affects the SSSP query performance. However, the total number of unnecessary columns can be estimated through η_{nulls} (see Equation 2).

Finally, we also carried a scalability analysis to identify the highest reduction achieved in the SSSP query processing time while varying: (i) the number of iterations performed over the DBLP graph; and (ii) the distinct parameter value used to generate the synthetic graphs. Figure 7 shows the largest reduction of the query processing time achieved over the DBLP dataset while varying number of iterations. In Figure 7a, the largest reduction was achieved with 2 iterations, and as the number of iterations increases, this reduction decreases. This implies that the query performance tends to degrade as the Edge-k and `result` tables are repeatedly joined. Nevertheless, solutions that employ too many joins are not usually well-suited for an RDBMS, such as looking for collaborations that have at least 5 authors. Considering our real dataset, performing more than 5 iterations means looking for collaboration paths between authors with lengths having at least 5 distinct authors. Such large lengths tend to be less frequent. Hence, it would not be particularly interesting to keep SSSP running any further than the number of iterations for which it already ran.



(a) DBLP. (b) Newman-Watts-Strogatz. (c) Erdos-Renyi.

Fig. 7: The largest query processing time reductions, using the Edge-k method. (a) Varying the number of iterations in the DBLP dataset; (b) Increasing numbers of vertices for Newman-Watts-Strogatz graph; and (c) Increasing the number of edges for Erdos-Renyi graph (see Table I).

Considering the synthetic datasets, increasing the number of vertices $|V|$ in the Newman-Watts-Strogatz graph (Figure 7b) and the probability p of edge creation (Figure 7c) have provided better processing times. Both Figures 7b and 7c show that our proposal provides even better results as the volume of data grows. That is, graphs that would have more rows in a conventional edge table (and, consequently, a larger number of repeated *source* vertex in the conventional implementation) would allow a larger reduction in both query processing time and dataset size.

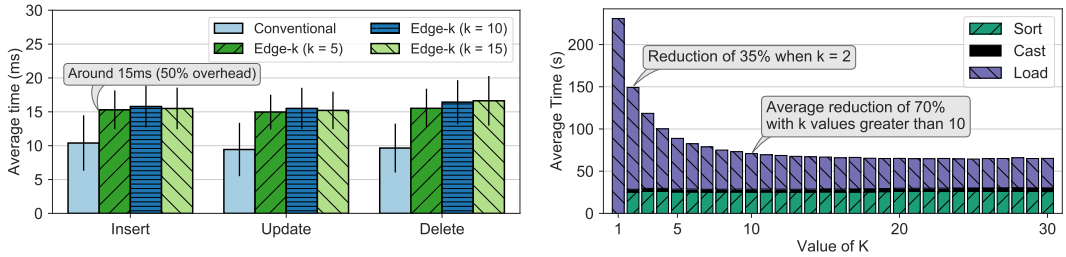
These experimental results show that it is possible to accomplish good performance improvements regardless of the number of vertices and of resulting vertex degrees. Considering the number of iterations performed by the SSSP procedure, the reduction in query processing time diminishes as the number of iterations increases. However, this is an issue already known in RDBMS-based solutions involving too many join operations.

4.3 Table Management Analysis

This section summarizes two analyses; the overhead in individual insert, update, and delete operations; and the bulkload operation. Both analyses are performed over the DBLP dataset. Regarding the first analysis, since the most significant reductions in dataset size and SSSP query time occurred for k varying from 1 to 15, we evaluated the overhead in the insert, update, and delete operations using $k = 1$, 5, 10, and 15. Notice that $k = 1$ is a special case where the Edge-k table corresponds to a conventional edge table. We started by randomly generating 1,000 new edges, inserting all of them into each one of the four tables; *i.e.* in the conventional edge table and in the Edge-k tables with $k = 5$, 10, and 15, respectively. Then, we updated the weight property of the new edges in every table.

Finally, we deleted the same 1,000 edges from each table. Figure 8 shows the average time and the standard deviations, calculated based on the time to manipulate the 1,000 random edges, discarding 10% of the shortest and 10% of the largest times.

Based on the times observed in Figure 8a, the overhead is about 50% for the insert, update, and delete operations when using the Edge-k table with $k \neq 1$. That is, instead of executing each of the three operations in 10 ms, they took approximately 15 ms. The elapsed times for $k = 5$, 10 or 15 were almost the same, indicating that the overhead occurs mostly due to the search for the desired *target* vertex. Since this search is performed in main memory — *i.e.* after loading the affected rows from the disk — the value of k does not pose a heavy impact on the processing time. However, there is an overlap among their standard deviations, especially in the insert operation. Nevertheless, most of the graph analyses are performed over static graphs, thus this overhead does not affect the final user. Even for dynamic data, many applications requires search operations more frequently than insert, update and delete operations. For those applications, our Edge-k method is better suited.



(a) General operations in an RDBMS.

(b) Bulkload average times per k value.

Fig. 8: (a) Average elapsed time of insert, update and delete operations in an RDBMS over a Edge-k table. (b) Average execution time of bulkload operations for varying k, which are performed following the three steps described in Section 3.2: Sort, Cast, and Load.

For the bulkload analysis, we followed the three steps described in Section 3.2: Sort, Cast, and Load. Figure 8b shows the average bulkload times as k varies from 1 to 30. Each measurement was collected 20 times, discarding 10% of the shortest and 10% of the largest times. Notice that for the conventional edge table (*i.e.* $k = 1$) only the Load step is required, although it requires more time to complete when compared to other values of k. By comparing the k values of 1 and 2, we observe a reduction of 35%, even executing the Sort and Cast steps (required when $k \geq 2$). As k increases, the time reduction stabilized around $k = 10$ with a reduction about 70% in total. The time reduction during bulkload operations are mainly related to the sharp reduction in the η_{rows} (see Equation 1).

4.4 Comparison of Edge-k to the Conventional Edge Table and a NoSQL Graph Database

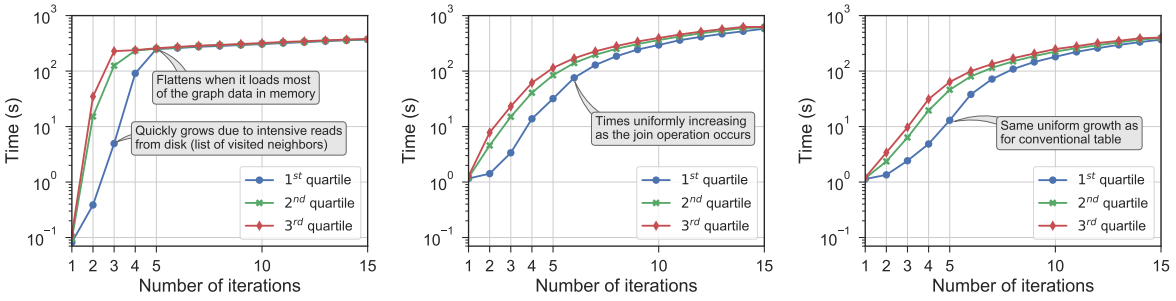
In this section, we compare Edge-k to the NoSQL graph database Neo4j, as well as to the conventional edge table. Considering the sharp reduction in the SSSP query processing time, which occurs for k up to 15, as discussed in the last two sections, this comparison was performed using $k = 10$. We also compared with the conventional edge table. We implemented both graph degree distribution and SSSP queries in Neo4j using the Java API, aiming at comparing the same algorithm in both databases.

The degree distribution query counts the number of vertices for each possible degree (*i.e.* the number of neighbors of each vertex) in the graph. In Neo4j, this query was implemented performing a loop through all vertices counting the number of neighbors for each vertex. In PostgreSQL, this query is performed with the `GROUP BY` clause on the *source* vertex identifier, followed by the `COUNT` aggregate function on the number of associated *target* vertices. Considering the Edge-k table, the query counts the number of *target* vertices in each row. The query finishes by summing the quantity of *target* vertices per row and employing the `GROUP BY` clause on the *source* vertex identifier. After

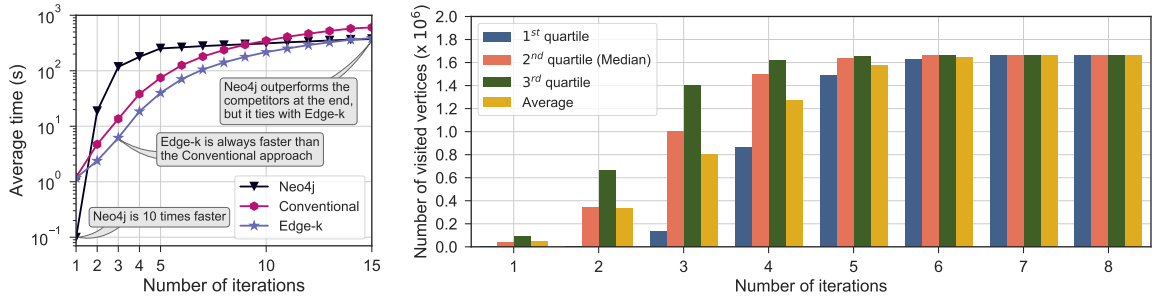
running the degree distribution query 30 times, and discarded 10% of the shortest and 10% of the largest times, we computed the average time for each evaluated approach. The result was **3.17s** for Edge-k, **6.21s** for Neo4j, and **9.68s** for the conventional edge table. This result is promising because, while Neo4j outperformed the conventional edge table, Edge-k outperformed both competitors.

We implemented the SSSP query on Neo4j using Algorithm 1, employing a HashMap structure for the temporary table. This algorithm enables comparing the DBMS-based approaches explored in this work. In this analysis, we focused on the evaluation of distinct initial *source* vertices (*i.e.* query centers) to compare Edge-k to Neo4j and to the conventional edge table. Such evaluation enabled analyzing distinct behaviors in terms of processing time as the query centers vary. The processing time changes because each *source* vertex can have either many or few connections, and being either a vertex close or remote with respect to the other vertices. Accordingly, a remote vertex is likely to demand less iterations to reach a hub and, until that point is reached, the query processing time will be low, since there will be fewer connected elements to be joined during the iterations of Algorithm 1.

Following that reasoning, we randomly chose twenty query centers, ten having vertex degree near the minimum vertex degree distribution, and ten having degree near the maximum degree distribution. For each query center, we calculated its SSSP varying the number of iterations from 1 to 15. At the 15^{th} iteration, SSSP stopped for some query centers because their `result` table did not change anymore (see line 7 in Algorithm 1). Each iteration was computed three times, in order to minimize any query processing time deviation. By grouping the twenty values based on their iteration number and sorting these values, we got all values that were in the first, second, and third quartiles. Figures 9a–9c show these values regarding Neo4j, the conventional edge table, and the Edge-k table, allowing us to analyze how much processing time changed for varying degrees.



(a) Neo4j processing time. (b) Conventional processing time. (c) Edge-k processing time.



(d) Average time in all systems. (e) Number of visited vertices per iteration.

Fig. 9: Time comparison for the SSSP query on Neo4j, on conventional edge table, and on Edge-k table, as the initial *source* vertex and the number of iterations vary. For each iteration, we also present the number of visited vertices.

For the three upper plots, the first quartile represents 25% of the elapsed time as the query center

varies, the second represents 50%, and the third represents 75%. Figure 9e shows the number of visited nodes per iteration considering the quartiles, including the respective average value. These results show that, in all cases, the series converge toward the same elapsed time. This means that the time is independent of the query center as the number of iterations increases. This behavior occurred for all evaluated techniques. The experiment evaluates the deviation from the median time of the three cases. We provided different perspectives to visualize the results, which reveal the inherent structure of the complex network representing the DBLP data. We showed that the results behave similarly for any query center, and also that our analysis is stable and that it can be generalized to any similar complex network, that is, to any free-scale network.

Additionally, Figure 9a shows that the execution times of Neo4j quickly grow for all quartiles. This is a consequence of intensive disk read operations, caused by the sharp increase in the visited neighbors list size (see Figure 9e). We always cleaned the cache before each SSSP query execution and employed a centralized version of Neo4j, thus the database demands a large processing time to bring the accessed vertices to the main memory. Such finding was expected according to Section 2, which relates the performance of Neo4j with the query result size. However, after loading most of the graph in the main memory, each iteration takes a small time. Considering the conventional and Edge-k tables, the elapsed time of both approaches increase uniformly.

Finally, Figure 9d shows the average time for the twenty query centers comparing Neo4j, conventional edge table and Edge-k table. In the first iteration, Neo4j was 10 times faster, but in the next iterations, it was negatively impacted by the visited neighbors list size, as previously discussed. In the 15^{th} iteration, Neo4j outperformed the competitors, but with an execution time very similar to our proposal. Furthermore, the Edge-k table always outperformed the conventional edge table for any number of iterations greater than 1.

5. CONCLUSION

The RDBMSs have been challenged by the growing volume and complexity of graphs demanded and generated by recent applications. Therefore, an RDBMS provides the infrastructure to manipulate such data, but its efficiency depends on the graph representation. Despite the several ways to represent graph data in an RDBMS, most of the frameworks that process graph queries focus on a single approach to store the edges of a graph. In this context, we proposed the Edge-k table, an alternative storage approach. It consists of representing the neighborhood of each vertex in a few or even in a single row. To maintain the robustness and to keep compatibility to the Relational Model, we defined that each group can store up to a predefined amount of k edges in a single row.

This work is based on the assumption that a Edge-k table can reduce the dataset size and, consequently, the processing time of queries. In fact, regarding query processing time, the experimental results over a real dataset showed a maximum of up to 66% at the second iteration of a SSSP query, as well as an average reduction of about 58% for the first four iterations. Regarding the two synthetic datasets evaluated, we obtained an average reduction ranging from 50% to 54%. Those results highlight the positive impact of Edge-k when compared to existing RDBMS-based graph-data management frameworks. We also performed three analyses to explore our method, described as follows.

- ◇ **Insert, update, and delete operations.** The overhead of a Edge-k table for such operations was about 50%, increasing the execution times to around 15 ms. However, since such operations usually are much less frequent than search operations, the overall impact of Edge-k is minimal;
- ◇ **Bulkload operations.** We proposed an method to perform bulk insertions into the Edge-k table that executes three steps (*i.e.* Sort, Cast, and Load). In the corresponding analysis, we achieved a time reduction average of 70% when $k \geq 10$, mainly due to the smaller number of affected rows;
- ◇ **A more extensive evaluation of data retrieval.** Along with the SSSP queries, we evaluated the performance of Edge-k in a degree distribution query. Additionally, we compared Edge-k both

with the conventional edge table in an RDBMS and with a NoSQL graph database. Regarding the vertex degree distribution query, Edge-k has outperformed both competitors. Considering the SSSP queries, Edge-k has outperformed Neo4j in most iterations (13 out of 15). Furthermore, our proposal proved to be superior than the conventional edge table for any number of iterations.

Although we evaluated Edge-k for the SSSP and degree distribution queries, many other graph applications can benefit from it. We mainly focused on frequent query types commonly implemented by graph data management frameworks over RDBMSs. Nonetheless, this work can be extended in several directions. For example, it can be extended to exploit other types of graph queries, especially those requiring properties other than a single weight measure. By analyzing different properties, the impact could be evaluated, *e.g.* in the case of additional *null* values in a Edge-k table. Additionally, in a future work we will explore further possibilities of how to use Edge-k foundations to develop an index structure. Finally, we intend to explore the edges connected to three or more vertices at a time, evaluating how Edge-k deals with multigraphs.

REFERENCES

Abadi, D. J., Marcus, A., Madden, S. R., and Hollenbach, K. Scalable semantic web data management using vertical partitioning. In *Proceedings of the 33rd International Conference on VLDB*. pp. 411–422, 2007.

Albahli, S. and Melton, A. Rdf data management: A survey of rdbms-based approaches. In *Proceedings of the 6th International Conference on Web Intelligence, Mining and Semantics*. ACM, New York, USA, pp. 31:1–31:4, 2016.

Barabási, A.-L. and Pósfai, M. *Network science*. Cambridge University Press, 2016.

Bornea, M. A., Dolby, J., Kementsietsidis, A., Srinivas, K., Dantressangle, P., Udrea, O., and Bhattacharjee, B. Building an efficient RDF store over a relational database. In *Proceedings of the 2013 SIGMOD*. ACM, New York, USA, pp. 121–132, 2013.

Chen, R. Managing massive graphs in relational DBMS. In *2013 International Conference on Big Data*. IEEE, Santa Clara, CA, USA, pp. 1–8, 2013.

Corbellini, A., Mateos, C., Zunino, A., Godoy, D., and Schiaffino, S. Persisting big-data: The NoSQL landscape. *Information Systems* 63 (Supplement C): 1–23, 2017.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to Algorithms, 3rd Edition*. The MIT Press, 2009.

Fan, J., Raj, A. G. S., and Patel, J. M. The case against specialized graph analytics engines. In *Proceeding of the 17th Conference on Innovative Data Systems Research CIDR*. Online Proceedings, Asilomar, CA, USA, pp. 1–10, 2015.

Gao, J., Jin, R., Zhou, J., Yu, J. X., Jiang, X., and Wang, T. Relational approach for shortest path discovery over large graphs. *PVLDB Endowment* 5 (4): 358–369, 2011.

Gao, J., Zhou, J., Yu, J. X., and Wang, T. Shortest path computing in relational DBMSs. *IEEE Transactions on Knowledge and Data Engineering* 26 (4): 997–1011, 2014.

Jindal, A., Madden, S., Castellanos, M., and Hsu, M. Graph analytics using vertica relational database. In *International Conference on Big Data*. IEEE, pp. 1191–1200, 2015.

Levandoski, J. J. and Mokbel, M. F. RDF data-centric storage. In *International Conference on Web Services*. IEEE, Los Angeles, USA, pp. 911–918, 2009.

Neumann, T. and Weikum, G. The RDF-3X engine for scalable management of RDF data. *The Very Large Data Bases Journal* 19 (1): 91–113, 2010.

Scabora, L., Oliveira, P. H., Kaster, D. S., Traina, A. J. M., and Traina-Jr., C. Relational graph data management on the edge: Grouping vertices' neighborhood with Edge-k. In *Proceedings of the 32nd Brazilian Symposium on Databases*. SBC, pp. 124–135, 2017.

Silva, D. N. R. D., Wehmuth, K., Osthoff, C., Appel, A. P., and Ziviani, A. Análise de desempenho de plataformas de processamento de grafos. In *31st Brazilian Symposium on Databases, SBBD*. SBC, Salvador, BH, Brasil, pp. 16–27, 2016.

Sun, W., Fokoue, A., Srinivas, K., Kementsietsidis, A., Hu, G., and Xie, G. Sqlgraph: An efficient relational-based property graph store. In *Proceedings of the SIGMOD*. ACM, New York, USA, pp. 1887–1901, 2015.

Vicknair, C., Macias, M., Zhao, Z., Nan, X., Chen, Y., and Wilkins, D. A comparison of a graph database and a relational database: A data provenance perspective. In *Proceedings of the 48th Annual Southeast Regional Conference*. ACM, New York, USA, pp. 42:1–42:6, 2010.

Vukotic, A., Watt, N., Abedrabbo, T., Fox, D., and Partner, J. *Neo4j in Action*. Manning Publications Co., Greenwich, CT, USA, 2014.