Relational graph data management on the edge: Grouping vertices' neighborhood with Edge-*k*

Lucas C. Scabora[1], **Paulo H. Oliveira**[1], **Daniel S. Kaster**[2], **Agma J. M. Traina**[1], **Caetano Traina-Jr**[1]

[1]Institute of Mathematics and Computer Sciences – University of São Paulo (USP)
São Carlos – SP – Brazil

[2]Department of Computer Science – University of Londrina (UEL)
Londrina – PR – Brazil

{lucascsb,pholiveira}@usp.br,dskaster@uel.br,{agma,caetano}@icmc.usp.br

Abstract. *As the amount of data represented as graph grows, several frameworks are employing relational databases to manage them. However, the existing solutions store graphs creating a row for each edge in an edge table. In this paper, we propose Edge-k, a novel storage approach that combines additional columns in the edges table, allowing to tune the number of edges stored in a single row by taking into account the overall neighborhood of the vertices, thus providing a better table organization. Compared to the existing approaches in the literature, experiments reveal that our proposal was able to reach a speedup of 66% over a representative real dataset and up to 57% in synthetic datasets when processing Single Source Shortest Path queries. Hence, our solution advances the state of the art in the context of graph data management within relational databases systems.*

1. Introduction

Complex networks are present everywhere, from communication infrastructures to social networks and urban street organization [Barabási and Pósfai 2016]. These networks are usually represented as graphs, in which the nodes are mapped as vertices and their relationships as edges. The amount of applications using graph structures to represent data has increased significantly. Many of them focus on analyzing the characteristics of the graphs, applying algorithms such as finding connected components, page rank and Single Source Shortest Paths (SSSP) [Silva et al. 2016].

For example, consider a graph composed of researchers and their respective publications. The relationship between two authors is defined by their common publications, *i.e.* one of them is the other's coauthor. Suppose the weight w of the edge corresponding to such relationship can be determined by $1 \div n_p$, where n_p is the number of shared publications between any pair of authors. In this case, $0 \leq w \leq 1$, and weights close to 0 indicate stronger relationships, whereas weights close to 1 indicate weaker relationships. In this scenario, consider a query searching for a specific researcher's collaborative network, which aims at identifying not only its direct coauthors, but also the coauthors of their coauthors and so on. To run such a query, which would allow the analysis of the researcher's influence on its study field, the SSSP algorithm can be employed. Given those characteristics of the weights, a weight close to 0 could also be seen as a short distance between authors, due to the large amount of shared publications. Conversely, a weight close to 1 could be seen as a long distance between them, due to the small amount of shared publications (or none at all). Hence, the SSSP algorithm can determine the strongest connections among the authors starting at a specified author, since it is achieved by identifying the minimal paths from that author to the other ones.

As the amount of applications grows, the volume of data also increases to the point that it does not fit in main memory, compelling them to focus on I/O operations in disks. Relational Database Management Systems (RDBMS) provide an infrastructure to support graph data management with some useful features such as storage, data buffering, index and optimizations. The main problem is that the Structured Query Language (SQL) is not adequate to express graph related queries. To solve this problem, a number of frameworks have been proposed. The FEM (Frontier, Expand and Merge) framework has been proposed to translate analytical graph queries into SQL commands [Gao et al. 2011, Gao et al. 2014]. This framework implements an SSSP query through tracking frontier vertices and iteratively expanding them, merging the new elements with the set of visited ones. The Grail framework [Fan et al. 2015], as well as its adaptation in 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 representation, not exploring the query execution performance in alternative representations. Both frameworks employ one relation for the set of edges and another for the set of vertices. The edges are organized as a list, each of them stored as a tuple in the table. The vertex degree (*i.e.* number of neighbors) is the number of edges incident to it, thus, in this representation, it determines the number of rows required to store the vertex.

In this paper we propose Edge-*k*, a novel and flexible strategy to store the neighborhood of a vertex in a RDBMS, grouping a predefined number of edges in the same entry. Particularly, since an entry is handled by the RDBMS as a table's row, we aim at reducing the overall number of rows required to store the graph while keeping the amount of *null* values in the edge table as small as possible, avoiding to generate wide tables and improving the performance of queries. Reducing the number of rows decreases the amount of data blocks used by the RDBMS as less space is dedicated to row headers, therefore minimizing the quantity of I/O operations. Furthermore, by grouping neighbor vertices in the same row, Edge-*k* ensures that at least part of them will be contiguously stored in disk, which can also contribute to achieve higher processing performance.

Regarding the aforementioned frameworks, we adapted the SSSP query to Edge-*k*, which was used to evaluate both real and synthetic datasets. This paper focuses on the SSSP algorithm because it is widely employed in many applications, such as discovering indirect relationships in a social network and finding minimal paths to interesting places in a city. The average query execution time revealed a strong correlation with the number of data blocks required to store the graph data. Thus, decreasing the amount of blocks, which is related to reducing both rows and *null* values, also speeds up queries. In the synthetic datasets, we achieved gains in query time from 46% to 57%. On the other hand, the real dataset allowed a gain of up to 66% regarding the first 2 iterations of the SSSP algorithm, whereas for 4 iterations the gain achieved 49%.

The remainder of this paper is organized as follows. Section 2 describes the related work, regarding existing approaches for graph data management using RDBMS. Section 3 details the proposed Edge-*k* storage approach. We provide an experimental analysis and discussion of our proposal in Section 4. Finally, Section 5 presents the conclusions.

2. Related Work

Existing storage schemas for graph data management employing RDBMS include: triple store schema; binary tables; and n-ary tables [Levandoski and Mokbel 2009]. The triple store schema organizes the data into a triple *<subject, property, object>*, in which *subject* refers to an entity instance (*e.g.* the identifier of a vertex or edge) and *property* represents an attribute of the entity, whose value is denoted by *object*. Considering an entity-relationship model, each attribute and relationship of an entity is described by *property* [Neumann and Weikum 2010]. Grouping each distinct *property* into a specific table corresponds to the binary representation, in which the table has only two columns: entity identifier and property value. Including other properties as new columns in the same table leads to the n-ary representation, and a special case in which all properties are assigned to the same table results in a property table [Levandoski and Mokbel 2009].

The disadvantage of placing every property into a distinct column is that it increases the data sparsity due to the heterogeneity in a entity set. In order to deal with this issue, DB2RDF [Bornea et al. 2013] proposed to aggregate sets of properties in the same column. DB2RDF defines vertex rows with a maximum of k properties, each represented by a pair of property and value columns. If there are more than k properties, they are split in several rows. This approach reduces the space required to store the dataset because it requires a smaller number of physical columns to store properties. SQLGraph [Sun et al. 2015] extended DB2RDF employing the same approach in non-relational databases. However, both works focused only on associating vertices with their properties, not dealing with relationships among vertices.

An alternative way to represent adjacency lists employs a column of array type [Chen 2013]. In this approach, each row in the table stores all the neighborhood of a vertex in a single column. Although this representation disrespects the first normal form, the authors state that it is well suited for sparse graphs since it avoids *null* values. However, such proposal focuses on scenarios in which all vertices have a small degree. Furthermore, the authors did not explore the scenario in which the row size is bigger than the data block size. The same idea of splitting rows adopted in DB2RDF and SQLGraph can be a solution here, but the insertion of new neighbors of a vertex needs yet to be analyzed.

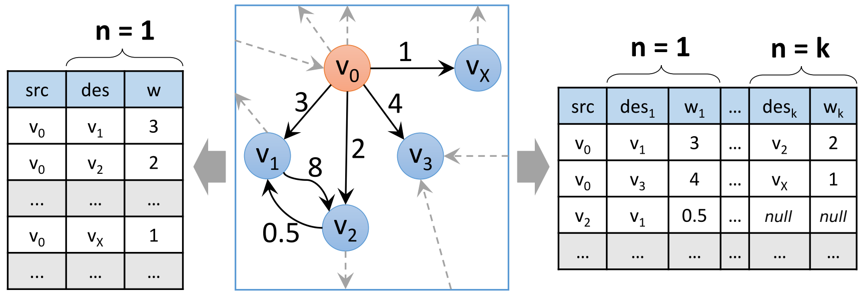
Our work draws inspiration also from the edge sharding approach introduced by the GraphChi framework [Kyrola et al. 2012], a disk-based system for computing graph algorithms efficiently. GraphChi splits its data into intervals, namely shards, in which the storage of edges is sorted by their source. The framework requires that each shard fits entirely in memory. However, GraphChi does not follow an RDBMS-based approach, which our proposal employs. Moreover, regarding a database system, these shards can be analogously seen as the data blocks in which the tables are stored.

Therefore, a similar approach can be used to improve RDBMS-based graph processing by grouping the neighborhood of a vertex in the edge table according to distinct values of k, which is a feature that neither DB2RDF nor SQLGraph provide for an RDBMS. This approach is the basis of Edge-*k*, which is presented in the next section.

3. The Edge-*k* Table Organization

A graph G is denoted as $G = \{V, E\}$, where V is a set of $|V|$ vertices and E is a set of $|E|$ edges, where both vertices and edges can have properties. Our work deals with the representation of edges. In the context of the Relational Model, on which our proposal is based on, edges and vertices are mapped into two distinct relations. Each vertex in V has a unique identifier attribute (*id*) and as many attributes as needed for the properties. An edge in E is defined as an ordered pair $\langle src, des \rangle$, where src is the vertex *source id* and des is the *destination* vertex *id*. Additional information for the edges can be expressed as extra attributes in the relation, allowing to further detail the relationships among the vertices. For example, an edge is defined as $\langle src, des, weight \rangle$ when storing the measure $weight$ of the relationship between src and des vertices.

Our novel technique Edge-k implements the vertex and edge relations as regular tables, and groups multiple *destinations* of a vertex in a same row in the edge table, up to a defined maximum amount of k *destinations* per row. Thus, a tuple in the edge table can store up to k edges starting from the same *source* vertex. If the edges have a property assigned, an additional column is included for each distinct *destination* vertex. Considering that every edge has a weight, the edge table definition becomes $E_k = \langle src, des_1, w_1, ..., des_k, w_k \rangle$, where w_i is the $weight$ between src and des_i, $1 \leq i \leq k$. This simple, but effective organization reduces the table overhead with tuple headers, as well as stores contiguously adjustable pieces of the vertex neighborhood, enabling to reduce storage space, number of disk accesses and query response time.



src	des	w
v_0	v_1	3
v_0	v_2	2
...	...	...
v_0	v_x	1
...	...	...

src	des_1	w_1	...	des_k	w_k
v_0	v_1	3	...	v_2	2
v_0	v_3	4	...	v_x	1
v_2	v_1	0.5	...	*null*	*null*
...	...	...	...	...	...

Figure 1. Comparison between the conventional and proposed approaches.

Figure 1 illustrates the proposed technique for a directed graph with a *source* vertex v_0 and a neighborhood of size n and each edge has a weigh w as an assigned property. The table at the left shows the conventional approach, which stores one *destination* per row. On the right is the proposed approach, which allows to store up to k *destinations* in the same row. For values of n smaller than k, all the neighborhood fits in the same row and the remaining values are filled with *null* values. If n and k are equal, which is the ideal case, a single completely occupied row contains the whole vertex neighborhood. For values of n larger than k, the neighborhood is stored in groups of size k. The last group may have less than k neighbors and thus the row is completed with *null* values. In the figure, the number of neighbors around vertex v_0 is divisible by k, fitting in two complete rows. On the other hand, the *source* vertex v_2 requires *null* values to complete its row.

By adopting an edge table that groups k *destination* vertices in a single row, we have a reduction of the number of rows at the cost of allowing *null* values. Therefore, the efficiency of our proposal is related to the parameter k, which is a function of the number of rows and the exceeding space of the edge table. Minimizing both quantities reduces the number of tuples required to the edge table be stored and, consequently, improves query performance. However, our proposal introduces an additional cost when performing individual update operations. This occurs because inserting an edge requires searching an empty destination vertex on the Edge-k table before inserting it. As such operations are less frequent than the search operations, its impact on the overall query load is small.

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

This section presents measures useful for the Edge-*k* method. They are calculated based on the value k chosen and on the edge attributes. In Equation 1, for each vertex v, the amount of rows is defined as the smallest integer value that is greater than its total neighborhood size ($degree_v$) divided by the defined value of k. Then, by summing the number of rows for each $v \in V$, we obtained the total amount (η_{rows}) of edge table's rows.

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

In Equation 2, $degree_v \bmod k$ obtains the number of remaining neighbors that do not fill a complete row in the edge table. By subtracting that number from k, and employing the modulo operator once again, we determine number of *null* values for each vertex. Finally, by summing them, the total amount of *null* values (η_{nulls}) is obtained.

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

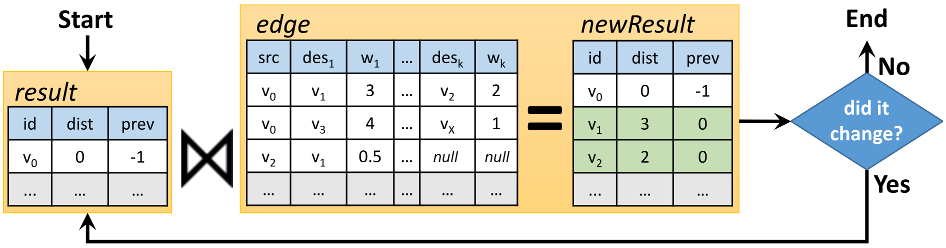
The optimal edge storage approach of a graph in an RDBMS table would use a single row to store the neighborhood of each vertex. In this case, both the row overhead imposed by the row header (which is necessary to store internal control data) and the space wasted due to *null* values are minimal. However, such organization would be properly represented only for the particular case in which every vertex has a fixed number of neighbors. This premise is not valid for most complex networks. As aforementioned, there are works that suggest employing RDBMS extensions, such as arrays. However, such approaches do not follow the Relational Model anymore, thus loosing robustness. Our work, on the other hand, strictly follows the Relational Model, employing a configurable control of the wasted space via the parameter k. The exceeding space ($exceeding$) occupied by the Edge-*k* table is given by Equation 3.

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

where $V_{neigh} \subseteq V$ is the subset of vertices of the graph containing a neighborhood and $size(v_{id})$, $overhead$ and $null\_size$ are, respectively, the sizes in bytes of the vertex *id*, of the row overhead and of the storage of a *null destination*. This equation considers space occupied by *null* values plus the wasted space occupied by vertices v_{id} without a neighborhood. The $null\_size$ depends on the implementation. If an RDBMS does not compact *null* values, it will reserve the standard size in bytes of the corresponding attribute. In such a case, $null\_size = size(v_{id})$. However, RDBMS usually apply techniques to save space when storing *null* values, and the real size varies according to the product.

3.2. Adaptation of Single Source Shortest Path (SSSP) procedure

A path $p = v_0 \overset{p}{\rightsquigarrow} v_n$ is a sequence of non-repeating edges $\langle v_0, v_1\rangle$, $\langle v_1, v_2\rangle, \ldots$, $\langle v_{n-1}, v_n\rangle$, in which there is one *source* vertex v_0 and one *destination* vertex v_n. A common additional attribute in the set of edges is a real-valued weight w, which defines a numerical measure between two adjacent edges in a path. The weight of a path is determined by the sum of the weigths of its constituent edges. The Single Source Shortest Path (SSSP) problem consists of finding the shortest path from a given source vertex v_0 to each vertex $v_i \in V$, that is, the paths $v_0 \overset{p}{\rightsquigarrow} v_i$ of minimal weight. To calculate SSSP in Edge-*k*, we adapted the corresponding procedure of the existing frameworks. Our implementation is illustrated in Figure 2. In short, the procedure iteratively joins a temporary table containing the weight of the current shortest path of every discovered vertex with the edge table, to visit new vertices and expand the frontier.



id	dist	prev
v_0	0	-1
...	...	...

src	des_1	w_1	...	des_k	w_k
v_0	v_1	3	...	v_2	2
v_0	v_3	4	...	v_X	1
v_2	v_1	0.5	...	*null*	*null*
...	...	...	...	...	...

id	dist	prev
v_0	0	-1
v_1	3	0
v_2	2	0
...	...	...

Figure 2. General idea of the SSSP procedure in our proposal.

The procedure is as follows. First, a temporary table result is created to register all visited vertices. For each of them, it keeps track of the unique identifier (*id*), the minimal distance to the path source vertex (*dist*) and the previous vertex (*prev*) in the current shortest path, allowing to identify which edges compound such path. The procedure starts by inserting the *source* vertex v_0 in table result, which has a distance of 0 and no previous vertex (*i.e.* -1). Thereafter, result is joined with the edge table to expand the neighborhood of result's vertices and update the shortest path of every destination vertex, producing a new temporary table (called table newResult). To update the distances in each iteration, the procedure sums the accumulated distance ($dist$ in table result) and the respective distance to the next neighboring vertex (recall that columns $w_1 \ldots w_k$ of the edge table store the distances from src to $des_1 \ldots des_k$ in edge E_k).

Regarding the example in Figure 2 (which corresponds to the graph of Figure 1), after first iteration the distances from v_0 to v_1 and from v_0 to v_2 are, respectively, (0+3) and $(0 + 2)$. The procedure aims at minimizing such distances, prioritizing the minimal paths discovered so far. Therefore, in second iteration, the weight of path $v_0 \overset{p}{\rightsquigarrow} v_1$ is updated to 2.5, through the expansion of vertex v_2, and so on. If tables result and newResult are equal, then all the shortest paths have already been found and the procedure finishes. Otherwise, table result is replaced by newResult and another iteration is executed.

Let's look at the major competitors of Edge-*k*, in which the key difference is in joining table result with the edge table. In the existing frameworks, the edge table stores only one destination per row. Therefore, when joining the tables result and edge, the procedure just executes an ordinary join operation. In Edge-*k*, on the other hand, for any value of k such join operation is accompanied by an unnesting operation, which consists of transforming the multiple $des_1 \dots des_k$ columns into separate rows.

4. Experimental Results

4.1. Setup

We evaluate our Edge-*k* method on both real and synthetic datasets, and due to space limitations we present the results on representative datasets. The real dataset was extracted from the *Digital Bibliography & Library Project*[1] (DBLP), containing information about authors and their publications. We modeled the set of authors as vertices and their relationships as edges, measuring the collaboration among them with a weight value. To analyze the impact of varying the number of vertices in the graph and the overall rate of neighboring vertices, we employed two synthetic graphs, generated through the Networkx[2] framework. The first one, called Newman-Watts-Strogatz dataset, models a small-world graph. Initially, it creates one ring including all $|V|$ 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, named 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 1 details the characteristics of all datasets, showing the number of vertices and edges, as well as the minimum (*min*), maximum (*max*) and average (*avg*) vertex *degrees*. For the Newman-Watts-Strogatz graphs, we fixed X to 200 shortcuts and varied the number of vertices $|V|$. Notice that their vertex degree size do not display a significant change. For the Erdos-Renyi graphs, we fixed $|V|$ and varied the probability p of edge creation with the values 1%, 3% and 5%. In this case, the vertex degree varies significantly, allowing the validation of more distinct scenarios. The experiments were run in the RDBMS PostgreSQL 9.5.6. We used a machine equipped with an Intel® Core™ i7-2600 @ 3.40GHz processor, 8GB of DDR3 1333MHz RAM memory, two SATA 6Gb/s 7200RPM hard disks set up in RAID 0, and Linux operating system Fedora release 25.

Table 1. Characteristics of the real and synthetic datasets.

Measure	DBLP	Newman-Watts-Strogatz			Erdos-Renyi		
		$X = 200$	$X = 200$	$X = 200$	$p = 1\%$	$p = 3\%$	$p = 5\%$
$\|V\|$	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
min *degree*	1	203	204	205	56	214	369
max *degree*	2100	245	238	247	126	334	536
avg *degree*	10.05	220.12	219.99	220.01	90.06	269.84	449.84

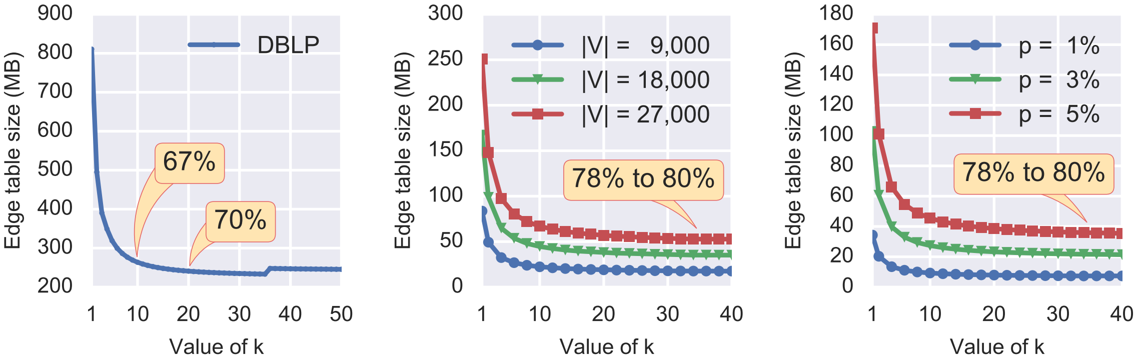
4.2. Dataset Size Reduction

This section presents the dataset size reduction achieved when employing Edge-*k* to group neighboring vertices in a same row in the edge table. Consequently, the overall amount

[1] Collected at March 16th, 2017 from `http://dblp.uni-trier.de/xml/`

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

of row overhead space is decreased, along with the number of times that each *source* vertex is stored. Figure 3 shows the size reduction achieved for each dataset. Considering the results for the DBLP dataset shown in Figure 3a, we observe a reduction of 67% at $k = 10$, this value of k coincides with the average degree as shown in Table 1. As k increases, the overall reduction stabilizes in approximately 70% ($k = 20$ and beyond).



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

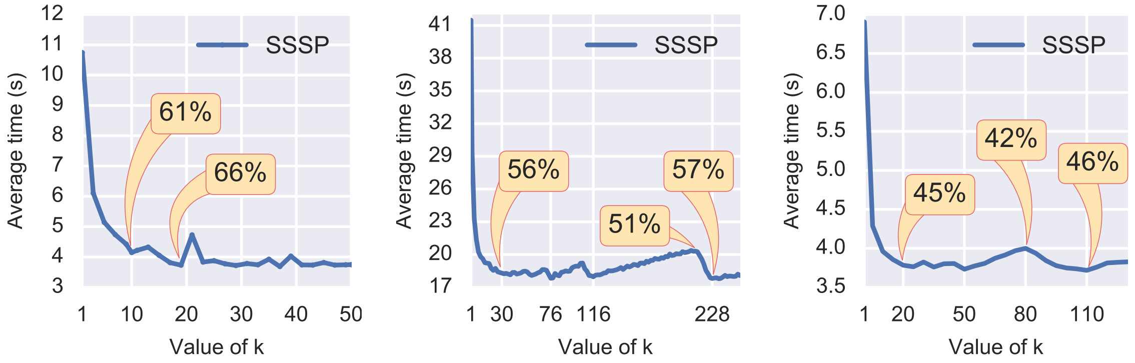
Figure 3. Size reduction of the edge table achieved in each dataset as *k* varies.

Regarding the synthetic datasets, both Figure 3b and 3c depict a similar behavior. Regardless of varying the number of vertices (in the Newman-Watts-Strogatz graph) or the vertex degree (in the Erdos-Renyi graph), the reduction achieved for $k \geq 30$ ranges from 78% to 80%. As explained by Equation 1, the sharp reduction in the dataset size was analogous to the decrease of η_{rows}, for the same range of k values. It is worth noticing the potential of our approach in terms of relation size reduction. This is especially evident when k varies from 1 to 10, where a drastic reduction occurs in the size of the edge table — achieving 67% already for the first values. For greater values of k, the neighbors of a vertex are able to be stored in less rows — most of them eventually fit in one row. In this scenario, as k grows, the number of *null* values stored in the edge table also increases, since the table becomes larger than required. As *null* values occupy an irrelevant space in the RDBMS PostgreSQL (apart from the bitmap structure used to keep track of *null* values, if any, in each row), having more *null* values does not harm the storage needs, which is why the size reduction remains stable even for greater values of k.

4.3. SSSP Query Processing Time

To analyze the impact of the dataset size reduction on queries, this section focuses on the SSSP query processing times. For each dataset, we defined a query whose starting point was the vertex having the highest degree — the largest number of neighbors. Each query was executed 30 times and had their execution time measured. Thereafter, we obtained the average execution time, after discarding 10% of the shortest and 10% of the longest times so as to rule out outliers. 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 and result are joined to compute the minimal paths, as described in Section 3.2. 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. For the synthetic datasets (from the starting point we defined), a maximum of 4 iterations was enough to visit all vertices in the graphs. Therefore, we executed the SSSP query on them without limiting the iterations.

Figure 4 presents the average SSSP query time when varying k for two iterations on the DBLP graph (Figure4a), for the Newman-Watts-Strogatz graph containing 27,000 vertices (Figure4b) and for the Erdos-Renyi graph with probability p = 1% (Figure4c). The query processing time reduction observed in Figure 4a achieved 61% for the average degree (*i.e.* for $k = 10$, as shown in Table 1), reaching 66% for $k = 20$ and remaining stable after this point. It is worth noticing that this behavior is analogous to the dataset size reduction. Figure 4b shows a similar behavior, presenting an initial reduction of 56% at $k = 30$ and stabilizing after this value with slight variations. Particularly, such variations range from 51% to 57% for values of k between 30 and 247, *i.e.* up to the maximum degree of the dataset, as presented in Table 1. This same behavior can be observed in Figure 4c, and in the other evaluated synthetic datasets, in which the reduction was of 45% for $k = 20$ and there were small variations ranging from 42% to 46% for values of k between 20 and 126. The observed peaks of Figures 4a to 4c correspond to the oscillations in the number of *nulls* (η_{nulls} from Equation 2) while evaluating different values of k.



(a) DBLP with 2 iterations.

(b) Newman-Watts-Strogatz with $|V| = 27,000$.

(c) Erdos-Renyi with $p = 1\%$.

Figure 4. SSSP query processing time in three scenarios for varying *k* in Edge-*k*.

According to Equation 1, the ideal cases occur when the overall degrees are divisible by k, which is when the average number of *null* values is smaller in the edge table. We can observe in Figure 4b that the best value between the minimum and maximum degrees is $k = 228$. However, that is not the single good value, since $k = 116$ and $k = 76$ also provide equivalent reductions in query processing time. The fact of several k values being appropriate is explained by the modulo operator in Equation 2: the number of *null* values rises — as does the query processing time — and falls at the next well-suited value of k — along with the query processing time as well —, by which the overall neighborhood is divisible. That is, minimizing both η_{rows} and η_{nulls} (Equations 1 to 3) is important when analyzing a proper value for k, which we empirically evaluated in the experiments.

Additional experiments not presented in this paper were performed for values of k greater than the maximum degree of the graph. However, we found that beyond this point the average query time only increases. This is because the amount of unnecessary columns in edge table keeps rising, adding more validations to the unnesting operation.

The adapted SSSP algorithm performs a loop through each one of the k *destination* vertices. Thus, assigning a value too large for k affects the overall performance.

As previously mentioned, the dataset size reduction is analogous to the average query processing time reduction, which indicates a correlation between them. To measure it, we calculated the Pearson coefficient between the amount of data blocks of edge table, while varying its k value, and the average elapsed time of SSSP queries, in which values closer to 1 imply a positive linear correlation. In the DBLP dataset, the coefficient obtained was 0.976 for 2 iterations and 0.827 for 5 iterations. In the synthetic datasets, the coefficient obtained was at least 0.943. Accordingly, reducing the dataset size (focusing on the edge table) also lowers the query processing time. This corresponds to the expected result, since reducing the dataset size should also decreases the amount of data blocks in the disk, minimizing the number of I/O operations and optimizing query processing.

4.4. Scalability Evaluation

We carried out another analysis to identify the highest reduction achieved in the SSSP query processing time varying: (i) the number of iterations performed over the DBLP graph; and (ii) the distinct parameter value used to generate the synthetic graphs. Figure 5 summarizes the highest query processing time reduction achieved over the DBLP dataset for each number of iterations employed. In the figure we can observe that the highest reduction has been achieved with 2 iterations, and as the number of iterations increases, the reduction decreases. This behavior shows that, as the tables edge and result are repeatedly joined, the query performance tends to degrade in both storage approaches. Nevertheless, solutions that employ too many joins are not usually well suited for RDBMS. For example, considering our real dataset, performing more than 5 iterations means looking for collaborations spanning at least 5 authors, which tends to be less frequent. Hence, in terms of a real application, it would not be particularly interesting to keep running the SSSP procedure any further than the number of iterations we already ran for.

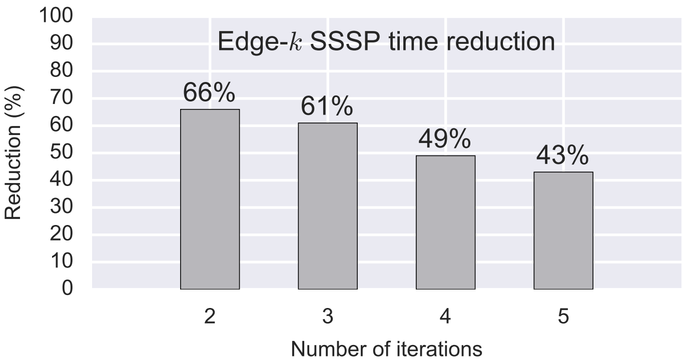
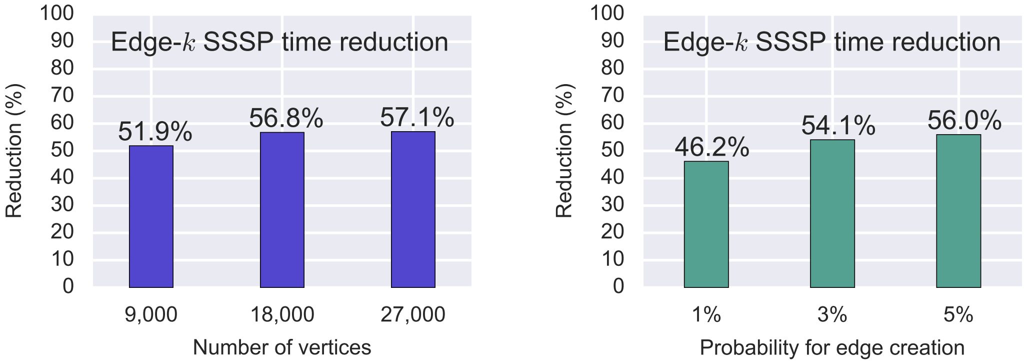


Figure 5. Highest query processing time reduction achieved, thought Edge-*k* method, for each number of iterations over the DBLP dataset.

Considering the synthetic datasets, Figure 6 shows the highest query processing time reductions according to the parameters considered. By increasing the number of vertices $|V|$ in the Newman-Watts-Strogatz graph, as shown in Figure 6a, query performance also increases, reaching up to 57% of query processing time reduction for the greatest number of vertices $|V|$ = 27,000. Specifically, such larger reduction was achieved when

k = 76, k = 116 or k = 228, as previously discussed regarding Figure 4b. Figure 6b depicts a similar behavior: by increasing the probability p of edge creation — which also changes the minimal, maximum and average degrees —, the corresponding query processing time reduction also increases. Both figures spot that our proposal provides even better results as the volume of data grows. That is, graphs that have more rows in the edge table (and, consequently, a higher amount of repeated *source* vertex in the conventional implementation) allow a higher reduction in both query processing time dataset size.



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

Figure 6. Highest query processing time reduction achieved, thought Edge-*k* method, for each parameter of Table 1 over the synthetic datasets.

The experimental results show that, regardless of the number of vertices and of resulting vertex degrees, it is possible to accomplish good performance improvements. Considering the number of iterations performed by the SSSP procedure, the reduction in query processing time decreases as the number of iterations increases, as already seen in various RBDMS-based solutions involving too many join operations. However, regarding the real scenario evaluated, an indirect relationship between two authors having more than three coauthors in the path joining them (*i.e.* requiring more than 5 iterations) is less likely to occur, thus irrelevant for our experimental analyses.

5. Conclusion

Graph data stored in RDBMS are growing in volume and quantity, requiring efficient management applications to properly deal with them. Despite the several ways to represent graph data in RDBMS, most of the frameworks focus on a single approach to store the edges of a graph. In this context, we have proposed Edge-*k*, a storage approach for the edge table of a graph, which consists of grouping at least part of the neighborhood of each vertex in a same row. This work has been conducted based on the assumption that this approach is able to reduce the dataset size and, consequently, the processing time of queries over the edge table. In fact, regarding query processing time, the experimental results show a reduction around 66% on real and 57% on synthetic datasets, which highlight the impact of our proposal. Although we have evaluated our storage approach for SSSP queries, many other graph applications benefit from it as well.

Due to space limitations, we focused on a frequent query type implemented by the general graph data management frameworks in relational databases. Nonetheless, this work can be extended in several directions. Firstly, a future work will explore the use of index structures in our storage approach, as well as different applications in other scenarios. Secondly, our approach could be extended to include edges connected to three or more vertices at a time, that is, to hypergraphs. Lastly, other databases not necessarily adopting the Relational Model (*i.e.* NoSQL databases) will be analyzed, and we will compare our proposal to the Neo4j NoSQL datastore.

Acknowledgments This research was supported, in part, by CAPES, CNPq and FAPESP. The authors would like to thank Gabriel Spadon and Willian D. Oliveira for their contribution on the paper and the server infrastructure provided by FAPESP (grant project 2014/26678-6).

References

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

Bornea, M. A., 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 SIGMOD*, pages 121–132, New York, NY, USA. ACM.

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

Fan, J., Raj, A. G. S., and Patel, J. M. (2015). The case against specialized graph analytics engines. In *Proceeding of the 2015 CIDR*, Asilomar, CA, USA. Online Proceedings.

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

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

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

Kyrola, A., Blelloch, G., and Guestrin, C. (2012). Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th Conference on Operating Systems Design and Implementation*, pages 31–46, Berkeley, CA, USA. USENIX Association.

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

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

Silva, D. N. R. D., Wehmuth, K., Osthoff, C., Appel, A. P., and Ziviani, A. (2016). Análise de desempenho de plataformas de processamento de grafos. In *31º Simpósio Brasileiro de Banco de Dados, SBBD*, pages 16–27, Salvador, BH, Brasil. SBC.

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