IEEE Access
Multidisciplinary ⁝ Rapid Review ⁝ Open Access Journal

Received November 23, 2020, accepted December 15, 2020, date of publication January 8, 2021, date of current version February 5, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3050081

An Efficient Algorithm to Count the Relations in a Range of Binary Relations Represented by a k^2-Tree

MARTITA MUÑOZ CANDIA [1], GILBERTO GUTIÉRREZ RETAMAL [1], AND RODRIGO TORRES-AVILÉS [2]

[1]Departamento de Ciencias de la Computación y Tecnologias de la Información, Universidad del Bío-Bío, Chillán 3780000, Chile
[2]Departamento de Sistemas de Información, Universidad del Bío-Bío, Concepción 4030000, Chile

Corresponding author: Martita Muñoz Candia (mmunocan@egresados.ubiobio.cl)

This work was supported by the University of Bío-Bío under Grant 192119 2/R, Grant 181315 4/R, and Grant 195119 GI/VC.

ABSTRACT Two sets A and B, whose elements fulfill a total order on operator $\leq$, can have a binary relation $R \subseteq A \times B$ represented by the k^2-tree compact data structure, which greatly improves storage space. Currently, *Count* query is managed by either using *Range* query or to modify the structure to have aggregate information, implying additional time or space in order to perform the query. This article presents *Compact Count*, which exploits the k^2-tree properties to reduce the paths to be scanned to count the numbers in a range r, thus ensuring an expected runtime of $O(\log_k r \log_k n)$ and storage of $O(\log_k r)$ with the k^2-tree parameters n and k. Our algorithm was compared through a series of experiments that consider both synthetic data with different distributions and real data, with a solution based on the *Range* algorithm. Experimental results show that *Compact Count* is 250 to 1,000 times faster than *Range* on synthetic and real data, respectively, with a small additional storage cost, as expected by the theoretical analysis.

INDEX TERMS Algorithm, compact data structures, counting query.

I. INTRODUCTION

Abinary relation R over two sets of objects A and B is a subset of the Cartesian product $A \times B$, that is, $R \subseteq A \times B$. There are many problems in the areas of mathematics, engineering, and computer science in which it is convenient to represent properties between objects by binary relations [1]. For example, $R \subseteq \mathbb{N} \times \mathbb{N}$ allows representing a set of points in a Cartesian plane (as Geographical Information Systems or GIS) or a binary relation $F \subseteq A \times A$ can represent the links between web pages in the World Wide Web, assuming A as a set of web pages. It is also possible to represent a communication network or transportation network by binary relations assuming that A is the set of nodes or cities, respectively. Binary relations can also be used to represent relative positions between points in a plane (point p is to the right of point q) [2] or to represent relationships between customers and products in recommendation systems [3]. Studying binary relations like this has been very fruitful, as researches were made to characterize them [4], to give new ad-hoc data structure [5], to compress them [6], to detect anomalies [7] and to model them [8].

A series of operations on sets of binary relations have been defined in the literature. For example, assuming that $x \in A$ and $y \in B$, some of these operations allow verifying if two objects are related, that is, if $(x, y) \in R$, or to obtain the set of objects related with a particular object $x \in A$, namely, $\{y \in B \mid (x, y) \in R\}$ (direct neighbors) or the set of all the objects related to an object $y \in B$, that is, $\{x \in A \mid (x, y) \in R\}$. Assuming a total order for both A and B and considering the operator $\leq$, a very important operation on a set of binary relations is *Range*, which allows obtaining all the relations (pairs $(x, y) \in R$) between objects specified in a range of objects from both A and B. Thus, considering intervals $[x_1, x_2]$, $[y_1, y_2]$ from A and B, respectively, *Range* calculates the set $\{(x, y) \mid (x, y) \in R \wedge x_1 \leq x \leq x_2 \wedge y_1 \leq y \leq y_2\}$ (for example, see [9]). A variant of this operation is *Count*, which allows counting the number of relations that exist in the specified range. We define *Count* more formally as

The associate editor coordinating the review of this manuscript and approving it for publication was Dongxiao Yu.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

Definition 1: Let $R \subseteq A \times B$, $[x_1, x_2]$ and $[y_1, y_2]$ be intervals from A and B, respectively, and range $r = [x_1, x_2] \times [y_1, y_2]$ and $count(r, R) = ||\{\{(x, y), (x, y) \in R \wedge x_1 \leq x \leq x_2 \wedge y_1 \leq y \leq y_2\}||$.

The *Count* query can be easily obtained from *Range*; however, one may want to obtain only the number of related elements and not the list of elements, especially in statistical studies. It is therefore important to find an efficient way to implement *Count*.

Binary relations have been traditionally represented by data structures such as graphs, trees, inverted lists, and binary matrices; adjacency lists and the adjacency matrix are the most commonly used to store and process them computationally [1]. Compact data structures have currently been proposed for storing large sets of binary relations. The data structure binary relation wavelet tree (BWRT), based on the wavelet tree [10] is proposed in [11] to represent binary relations. The k^2-tree data structure that efficiently stores (significant saving of memory) binary relations represented by an adjacency matrix is proposed in [12]. Unlike compressed data structures, compact data structures allow information to be directly processed in its compressed state without decomposing [13], which makes them especially suited for use in devices with limited capacity, such as smartphones and laptops, and/or benefit from the memory hierarchy. By requiring minimal storage it is possible to store the structure in memory types closer to the CPU (central processing unit) and thus improve application performance.

The objective of this research is to enhance the capacity of k^2-tree to efficiently manage the aggregate query *Count* without modifying the structure.

Although there exists considerable literature about the capacity (operations or queries) of the k^2-tree data structure (as in [12], [14], [15]), there is no efficient algorithm to calculate *Count* without modifying the structure. One way to directly calculate *Count* is by the *Range* operation (as suggested in [13]), but, this requires unnecessary access to several k^2-tree nodes, which implies an inefficient operation. Other solution is to incorporate additional information to the upper part of the conceptual tree, called k^2-treap [16], effectively having a trade-off between memory and speed. This solution only requires to identify the involved submatrices in the range, as the number of points in those submatrices are, in general, stored. This article proposes: i) an ad-hoc algorithm (*Compact Count*) to calculate *Count* from binary relations represented by a k^2-tree without additional information, and that benefits from the properties of the structure and ii) a series of experiments that evaluates the performance of *Compact Count*.

There are several algorithms that can benefit from *Compact Count*. For example, assuming that $R \subseteq \mathbb{N}^2$ is represented by a k^2-tree, a primary example is the calculation of the $K \geq 1$ nearest neighbors (Increasing Radius method [1]). The number of points in square C with edge size d and circumscribing the circumference can be calculated from a circumference centered on q and radius d. If the number of points in C is less than K, the same procedure can be applied with a radius of $2d$. Algorithms for solving the constrained skyline problem can also benefit [17], which is a variant of the skyline problem that requires ranking the skyline points according to the number of dominated points. But not necessarily we need additional algorithms. The *Count* query is useful by itself. It is well known that aggregate queries, like *Count* can be useful to give additional information for decision making process [18], or to count the number of links in or linked to a particular Web Page to ranking it. In particular, counting relations in k^2-trees are very important, as, in both scenarios, the information is not just sparse, but also clustered (by Tobler's first law of geography [19] and by alphabetical order in URLs [6]).

The remainder of the article is organized as follows. Section II describes the k^2-tree compact data structure and previous work over aggregate queries. Our algorithm is described in Section III. The theoretical analysis of our algorithm in terms of time and storage is presented in Section IV. Section V illustrates a set of experiments that show the behavior of the algorithm. Finally, the conclusions are discussed in Section VI.

II. BACKGROUND

The research done in this manuscript presents a way to efficiently count binary relations represented in a compact data structure called *k^2-tree*. Let us introduce this concept and the way that *Count* query has been addressed in it.

A. k^2-TREE

Let us consider a binary relation represented by a binary adjacency matrix M of size $n \times n$, such that n is a power of k. A k^2-tree is a tree representation of this matrix M in a compact way: M is subdivided in k^2 submatrices of $\frac{n}{k} \times \frac{n}{k}$. Then, each k^2 submatrix is represented by a bit whose value is 1 if at least one cell contains a 1 and 0 otherwise (all cells are 0). Those submatrices represented by 1 are recursively subdivided in k^2 submatrices. The subdivision ends when a submatrix is found with all its cells at 0 or when the individual cells are reached. Fig. 1 illustrates an example of an adjacency matrix and its respective k^2-tree representation where $k = 2$ and $n = 4$.

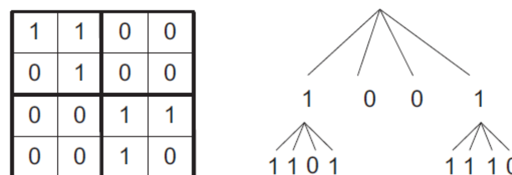

FIGURE 1. A matrix (left) and its k^2-tree representation (right). This k^2-tree is stored in bitmaps $T = 1001$ and $L = 11011110$.

Nevertheless, the tree is not stored as is. A k^2-tree is actually stored by two bitmaps called T and L. The internal nodes whose order is established by a widthwise path are stored in T, while leaf nodes ordered after a widthwise path are stored in L. The values stored in the bitmaps in Fig. 1 are $T = 1001$ and $L = 11011110$. This separation is performed

because L does not have the additional structures to respond to queries inherent to the compact data structures described below because it is unnecessary [20].

Any operation over a k^2-tree requires two basic operations over the bitmaps, which are typical of compact data structures. Let B be a bitmap of size l and the operation $rank_b(B, i)$ counts the number of bits of B whose value is b (zero or one) to position i with $1 \leq i \leq l$. In addition, the operation $select_b(B, i)$ calculates the position of B to where there are i bits with value b [13], [20]. To some extent, *select* is the inverse function of *rank* expressed as $rank_b(B, select_b(B, i)) = i$.

The operation *rank* can be calculated in time $O(1)$ and *select* in time $O(\log \log n)$ using storage $n+o(n)$ [20]. Storage is used for both storing the bitmap and the additional structure that reduces the response time of both queries. In the case of a k^2-tree, and to reduce the memory being used, this structure is only implemented over T, which performs *rank* and *select* queries over this bitmap.

It is possible to navigate the k^2-tree using the *rank* operation to find the ith child of a node inside the tree. In order to find the position of the ith child in bitmap $T : L$ of a node representing position x in bitmap T of the k^2-tree, equation (1) is applied as:

$$child_{i-1}(x) = rank_1(T, x) \cdot k^2 + i - 1. \tag{1}$$

For example, for k^2-tree in Fig. 1, if we want to obtain the second child of node 3, we have that $child_1(3) = rank_1(T, 3) \cdot 4 + 1 = 2 \cdot 4 + 1 = 9$. As bitmap T as only 4 positions, the second child of node 4 is in position $9 - 4 = 5$ of bitmap L.

B. PREVIOUS WORK

Let us, again, assume a binary relation represented by a binary adjacency matrix M of size $n \times n$, such that n is a power of k. The *Count* query returns the amount of binary relations (1 in the matrix) in a particular range $r = [x_1, x_2] \times [y_1, y_2]$.

Aggregate queries as *Count* have been considered in two dimensional grids and, in particular, for binary relations. Let us suppose that we need to count m relations. For example, in *wavelet trees*, which are a compact way to represents grids, *Count* query can be resolve in $O(\log m)$ time. The algorithm involves traversing the wavelet tree. It is possible to consider a time of $O(\frac{\log n}{\log \log n})$ using a modification of this structure [21].

Now, in the structure considered in this research, as explained in [13], the standard (and naïve) way to compute *Count* in k^2-tree is to use *Range* query (*i.e.* traversing the k^2-tree). This implies to retrieve all the relations m inside the range r, which has a worst-case scenario of $O(|r| + log_k n)$ and average of $O(\sqrt{m})$ [12].

An interesting alternative was given in [16]. In that work, Brisaboa *et al.* worked with two-dimensional geographic information (or OLAP cubes). In particular, the information managed can be seen as a two dimensional grid with values in the range $[0, d - 1]$. A compact data structure called k^2-treap is presented, where the topology of the positive values is stored in a k^2-tree, and additional information (as the positive values of the leaves and aggregate data in the internal nodes) are stored separately. Again, the base structure is modified in order to obtain better results in aggregate queries.

In practice, every upper node in the tree stores information for the *Sum* and *Count* query, but in an efficient way (storing just the difference between the current node and its parent).

In the experimental section, it is shown that storing aggregate information in the first 8 to 16 levels of the tree significantly improves the speed of *Count* query, but permanently sacrificing store space (below 30%).

In conclusion, solutions to aggregate queries have implied modifying the original structure. As our objective is to speed up *Count* query in k^2-tree without adding additional information to the compact data structure, this solution is out of our scope.

III. PROPOSED ALGORITHM: *Compact Count*

In this section, we describe our algorithm (*Compact Count*) to calculate the number of elements (relations or cells at 1) within a query range $r = [x_1, x_2] \times [y_1, y_2]$ over a set of stored data in a k^2-tree. The algorithm exploits the properties of the k^2-tree to discard as quickly as possible zones of the adjacency matrix that do not intersect with range r, which reduces the number of accesses to the k^2-tree nodes.

One of the properties of the k^2-tree that exploits the algorithm is established in the following lemma:

Lemma 1: Leaves of any k^2-tree node, which represent 1×1 cells, are adjacent to each other within bitmap L.

Proof: As previously indicated, the k^2-tree nodes are represented within bitmap $T : L$ (bitmap T concatenated with bitmap L) by performing a widthwise path through the tree. This implies that the leaf nodes of any submatrix are represented in adjacent locations within bitmap $T : L$. Therefore, the leaf nodes (1×1 cells) are represented contiguously within bitmap L. ■

Due to Lemma 1, it is possible to count the relations within a submatrix of the k^2-tree. The procedure implies to identify in which position in L the first and last descendant is found and obtain the number of 1s between these two positions by the $rank_1$ operation. For example, to count the number of 1s in the second non-empty submatrix ($[2, 3] \times [2, 3]$) of Fig. 1, the k^2-tree is descended until the fifth and eighth bit of L is reached.

This property of the k^2-tree is used by *Compact Count* to identify what we call *maximum quadrants* that constitute the range and to count the relations that each of them contains, as previously described.

Definition 2: A maximum quadrant for range r is a squared submatrix of the k^2-tree that is completely contained within r, but its parent node does not meet this condition.

In Fig. 2, for range $r = [0, 6] \times [0, 5]$ of the query (highlighted in yellow), submatrix $s = [0, 3] \times [0, 3]$ is a maximum quadrant. It should be noted that s is completely contained in r; however, the matrix on the level immediately above (matrix $[0, 7] \times [0, 7]$) of which s is one of the k^2 children is

IEEE Access

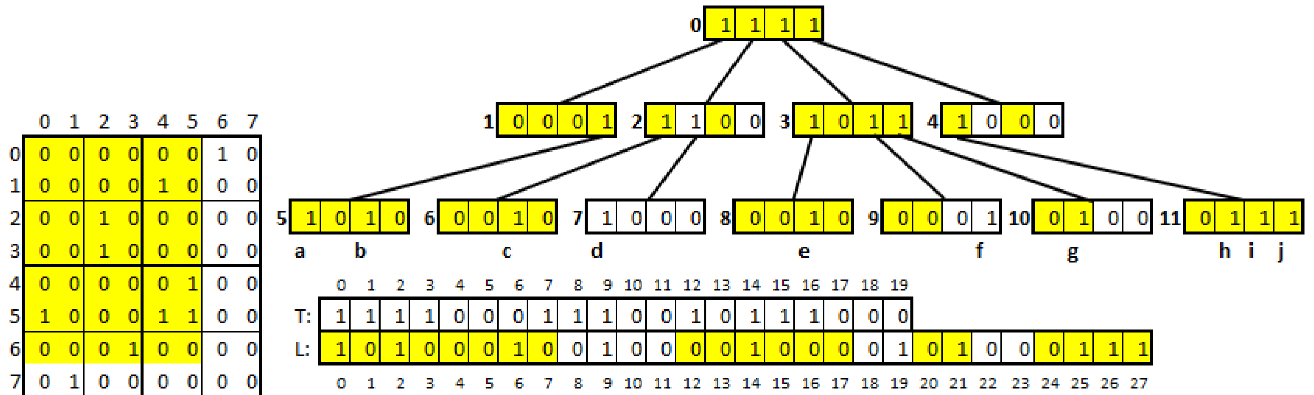

FIGURE 2. Example of ***Compact Count*** execution.

not a maximum quadrant. Similarly, submatrix $[0, 1] \times [4, 5]$ is a maximum quadrant, but submatrix $[0, 3] \times [4, 7]$ is not.

Therefore, let $|M|$ be the number of maximum quadrants that comprise range r.

Lemma 2: The number of relations in a range is the sum of the number of relations in each maximum quadrant that comprises it.

Proof: A range is comprised by one or more maximum quadrants. Given that the maximum quadrants are quadrants of the k^2-tree, the relations contained in them do not intersect each other. Therefore, the sum of the number of relations that each maximum quadrant contains is the total relations of the range. ■

Let p be the number of relations contained within r and m_i is the number of relations within the ith maximum quadrant that comprises it. Since r is comprised by $|M|$ maximum quadrants, it can be determined that:

$$p = \sum_{i=1}^{|M|} m_i. \tag{2}$$

This last statement is clear, as in Fig. 2, we have 12 maximum quadrants: $s_1 = [0, 3] \times [0, 3]$, $s_2 = [0, 1] \times [4, 5]$, $s_3 = [2, 3] \times [4, 5]$, $s_4 = [4, 5] \times [0, 1]$, $s_5 = [4, 5] \times [2, 3]$, $s_6 = [4, 5] \times [4, 5]$ plus the individual relations $[j, 6]$, with $j \in \{0, \ldots, 5\}$. In this case, we have $p = 2+1+0+1+0+3+0+0+0+1+0+0 = 8$.

A. HOW TO COUNT RELATIONS WITHIN A QUADRANT

Let N be a submatrix or quadrant represented by an internal node of the k^2-tree and indexed by j in bitmap T. The objective of this algorithm is to calculate the number of 1s in N.

Equation (1) describes how to navigate in the k^2-tree using bitmap $T : L$, descending to search for the location of the ith child. This is how to descend from an N submatrix to the first and last descendent leaf within L by iteratively using (1) to locate the first child node ($child_0(j)$) and the last child node ($child_{k^2-1}(j)$).

The problem with (1) is applying it over an empty node, which is a node that represents a submatrix with all its cells at zero; for example, $[0, 1] \times [2, 3]$ in Fig. 1. Therefore, it is important to not only identify the first/last descendant for each navigated level in the k^2-tree but also to locate the first/last **non-empty** descendant for each level.

Therefore, if $T : L[child_0(j)] = 0$, we can actually get the first non-empty child of a node located within $T : L$ in position j considering:

$$firstChild(j) = select_1(T, rank_1(T, child_0(j)) + 1) \tag{3}$$

The *rank* function over the location of the first child indicates the number of 1s existing before locating the first child. To locate the first non-empty child, a unit must be added to the result of the *rank* function and then the *select* function must be applied to obtain the searched location.

For example, in Fig. 2, if we need to get the first non-empty child of node 0, first we consider $child_0(0) = rank_1(T, 0) \cdot 4 + 0 = 4$. But as $T[4] = 0$, then we need to use $firstchild(0) = select_1(T, rank_1(T, child_0(0)) + 1) = select_1(T, rank_1(T, 4) + 1) = select_1(T, 4 + 1) = 7$.

In the same way, it is possible to obtain the position of the last non-empty child of a node located within $T : L$ in position j:

$$lastChild(j) = select_1(T, rank_1(T, child_{k^2-1}(j))) \tag{4}$$

The *rank* function over the location of the last child indicates the number of 1s existing before locating the last child. It is possible to obtain the location of the last non-empty child after applying the *select* function.

For example, in Fig. 2, the last non-empty child of node 3 is $lastChild(3) = select_1(T, rank_1(T, child_3(3))) = select_1(T, rank_1(T, 19)) = select_1(T, 9) = 16$.

Algorithm 1 presents the *getN* function, which counts the relations within a submatrix represented within bitmaps $T : L$ in position j. Overall, after verifying that the submatrix is neither leaf nor empty, the algorithm iteratively descends to identify the first and last non-empty descendant for each level of the k^2-tree until it reaches the leaf level. Finally, in line 19, the difference of *rank* between the last and first position is

applied. This operation determines the value of the number of 1s (relations) existing within the submatrix.

As previously explained in Subsection II-A, bitmap L has not additional structures to respond *rank* queries. This implies that to execute line 19, the additional structures must be extended to allow the application of the *rank* operation over L in constant time.

B. HOW TO COUNT RELATIONS WITHIN A RANGE

As explained, to count the relations within a query range in a k^2-tree indexed matrix, the maximum quadrants must be first identified and then the relations counted using *getN* over each of them.

Algorithm 1 getN($T:L, j$)

Input: $T:L$ bitmap that represents k^2-tree internal nodes
Input: j position in $T:L$ that represents the submatrix
Output: number of relations inside the submatrix

```
1: if j is a leaf then
2:    return L[j]
3: end if
4: if T[j] = 0 then
5:    return 0
6: end if
7: firstChild = j
8: lastChild = j
9: while firstChild is not a leaf and lastChild is not a leaf
   do
10:   firstChild = child_0(firstChild)
11:   if firstChild is not a leaf then
12:      firstChild = select_1(T, rank_1(T, firstChild) + 1)
13:   end if
14:   lastChild = child_{k^2-1}(lastChild)
15:   if lastChild is not a leaf then
16:      lastChild = select_1(T, rank_1(T, lastChild))
17:   end if
18: end while
19: return rank_1(T : L, lastChild) − rank_1(T :
    L, firstChild)
```

To identify the maximum quadrants, a queue is used to store those submatrices that intersect with both range r of the query and its complement in order to examine their children. The first submatrix to enter is the root, which represents the complete matrix. Each extracted submatrix N is subdivided into its k^2 children N_i, $1 \leq i \leq k^2$, which are verified for their relation to r to define the action to be performed with N_i:

- $N_i \subseteq r$ (N_i is contained in range r): *getN* is applied over N_i and 1s are added in N_i to the total of 1s in r.
- $(N_i \neq\subseteq r) \wedge (N_i \cap r \neq \emptyset)$ (N_i intersects r without being contained therein): N_i enters the queue.
- $N_i \cap r = \emptyset$ (N_i does not intersect r): N_i is discarded.

The algorithm ends when the queue is empty.

The following data are saved for each stored submatrix: $\langle (x, y), l, i\ \rangle$ where (x, y) are the coordinates of its corner nearest to the origin, l is the size of its edge, and i is the index that represents the submatrix in bitmap $T:L$.

Algorithm 2 CompactCount(k^2 tree k, Range r)

Input: k k^2 tree
Input: r query range
Output: number of relations inside r

```
1: total = 0
2: queue = ∅ {Process queue}
3: queue = queue ∪ {k.root} {An object ⟨(0, 0), k.n, −1⟩ is
   added to the queue queue}
4: while queue ≠ ∅ do
5:    N = queue.top
6:    for each child N_i in N do
7:       if N_i ≠ ∅ then
8:          if N_i ⊆ r then
9:             total = total + getN(k.T : L, N_i.x)
10:         else
11:            if N_i ∩ r ≠ ∅ then
12:               queue = queue ∪ N_i {The object
                  ⟨(x_{N_i}, y_{N_i}), l_{N_i}, i_{N_i} isadded⟩}
13:            end if
14:         end if
15:      end if
16:   end for
17: end while
18: return total
```

The *Compact Count* pseudocode is presented in Algorithm 2. Whether or not N_i intersects r is confirmed in line 7. If it does not intersect, it is immediately discarded. In the case that it does intersect, it is revised to determine if N_i is completely contained in r (line 8) or not (line 11). In the first case, *getN* is applied, while N_i enters in the queue in the second case.

C. AN EXAMPLE

Fig. 2 illustrates an example of an 8×8 matrix in which a range of $r = [0, 5] \times [0, 6]$ is delimited within it. Its representation in a k^2-tree and implementation in its respective bitmaps T and L also appear. Marked spaces in both representations show the path the algorithm must follow to find the number of relations that exist within the range.

TABLE 1. Execution of *Compact Count* over the k^2-tree of Fig. 2.

Action	Queue content	total
Start	$\langle (0,0), 8, -1\rangle$	0
Expand the root	$\langle (4,0), 4, 1\rangle, \langle (0,4), 4, 2\rangle, \langle (4,4), 4, 3\rangle$	2
Expand 2	$\langle (0,4), 4, 2\rangle, \langle (4,4), 4, 3\rangle$	3
Expand 3	$\langle (4,4), 4, 3\rangle, \langle (0,6), 2, 14\rangle, \langle (2,6), 2, 15\rangle$	4
Expand 4	$\langle (0,6), 2, 14\rangle, \langle (2,6), 215\rangle$	7
Expand 9	$\langle (2,6), 2, 15\rangle$	7
Expand 10	$\emptyset$	8

Table 1 shows the steps performed by the algorithm to count the 1s within the range marked in Fig. 2. For this

example, the nodes that enter the queue are represented using the same format described in Section III-B.

The root is initially inserted in the queue. After revising its children, node 1 is completely within the range; therefore, the *getN* function is applied to obtain the number of 1s, two in this case. The other three submatrices enter the queue.

There are two children in node 2 that are completely within the range and two outside. Of the two within, one is empty and the other (node 6) adds a 1 to the total. *getN* defines whether the node is empty or not.

When the algorithm revises node 3, two nodes are completely within the range and two intersect it. One of the nodes completely contained within the range is empty. The other (node 8) adds another 1 to the total of 1s. Nodes 9 and 10 enter in the queue.

As for node 4, one node is completely contained within the range and another intersects it. The node that is completely contained adds three 1s to the total, while the intersecting node is empty.

When revising nodes 9 and 10, node 9 does not contribute any new relation (1s) to the total and node 10 contributes one for a total of eight relations.

IV. ANALYSIS

A. TEMPORAL ANALYSIS

The cost of counting the number of relations (1s) within a submatrix N defined in the k^2-tree must be determined at the start. In this case, cost is measured on the basis of the number of *rank* and *select* operations performed.

Lemma 3: Counting the number of relations (1s) within a submatrix N is $O(h)$, where h is the height of the submatrix.

Proof: The actions of accessing both the first and last non-empty child node is $O(1)$ because it involves at most three *rank*/*select* operations for each one. These operations must be repeated h times for each level of the tree to descend the k^2-tree to the leaves. This is equivalent to a cost of $6h = O(h)$. ■

The temporal cost of obtaining the number of relations (1s) within a range r is defined by the sum of the cost of identifying the maximum quadrants (T_{getM}) and the cost of counting the 1s within these quadrants (T_{countM}) is

$$T = T_{getM} + T_{countM} \tag{5}$$

Given Lemmas 3 and (2), it can be concluded that the cost of counting the 1s within each of the maximum quadrants constituting range r is

$$T_{countM} = 6\sum_{i=1}^{|M|} h_i, \tag{6}$$

where h_i is the height of the ith maximum quadrant ($0 < h_i \leq H$ and H is the height of the k^2-tree).

To identify all the maximum quadrants, it is necessary to descend from the root of the k^2-tree to where each of the $|M|$ maximum quadrants is located. For each maximum quadrant N_i, $H - h_i$ levels must be traversed from the root to height h_i of each quadrant. Therefore, the cost of identifying all the maximum quadrants is

$$T_{getM} = \sum_{i=1}^{|M|}(H - h_i) = |M|H - \sum_{i=1}^{|M|} h_i. \tag{7}$$

Considering (5), (6) and (7), we obtain that

$$T = |M|H - \sum_{i=1}^{|M|} h_i + 6\sum_{i=1}^{|M|} h_i = |M|H + 5\sum_{i=1}^{|M|} h_i. \tag{8}$$

Recalling that $h_i \leq H$, it can be concluded that $\sum_{i=1}^{|M|} h_i \leq \sum_{i=1}^{|M|} H = |M|H$. In addition, $H = \log_k n$, where n is the matrix size, and applying this to (8), the temporal cost of *Compact Count* is limited by

$$T(n) \in O(|M| \log_k n). \tag{9}$$

The temporal cost of *Compact Count* depends on the height of the k^2-tree (thus the size of the matrix), and mainly on the number of maximum quadrants that constitute the range. Therefore, the temporal cost does not directly depend on the number of relations (1s) in the range but on their distribution.

When the empty maximum quadrants are larger and more numerous, the temporal efficiency of *Compact Count* is better.

B. SPATIAL ANALYSIS

Additional storage (not including storage occupied by the k^2-tree) that is required by *Compact Count* is determined by the storage occupied by the queue to maintain the submatrices that intersect r but are not completely contained in r. The quadrants maintained in the queue are the ancestors of the maximum quadrants. At any level, submatrices are those without non-empty children. For each submatrix removed from the queue, it will reinsert at least one submatrix provided it also has a maximum quadrant descendant. The re-entry of submatrices in the queue can only reach the parents of the maximum quadrants since, by definition, maximum quadrants do not enter in the queue. Assuming that the maximum quadrants $|M|$ that constitute the range come from different parents, it can be concluded that the queue can have at most $|M|$ elements.

This value can decrease because there can be maximum quadrants that are siblings and maximum quadrants whose ancestors were empty.

It is also important to consider the additional space required by the k^2-tree to respond to the *rank* query over bitmap L. This requires space of order $o(n)$, which is small compared with the queue.

C. BEST CASE

The key variable that determines both the temporal and spatial cost is the number of maximum quadrants $|M|$ within range r. Therefore, the best case minimizes this value. The lowest value for $|M|$ is 1. This occurs when r is a submatrix of the k^2-tree, regardless of its size. If r covers a quadrant equivalent

to $\frac{1}{k^2}$ of the matrix, the cost will be similar to when r covers only a 1×1 quadrant.

The temporal cost in this best case scenario is $O(\log_k n)$, which represents the descent from the root to identify the maximum quadrant and the subsequent descent to the leaves to identify the first and last descendant to finally perform the count.

In the best case of spatial cost, there will only be one submatrix for each iteration that meets the condition to enter the queue: the ancestor of the maximum quadrant. For each descended level until reaching the particular quadrant, only one submatrix will be within the queue. Therefore, spatial cost is $O(1)$.

D. WORST CASE

As explained in the previous subsection, the key variable that determines both the temporal and spatial cost is the number of $|M|$ maximum quadrants. Therefore, the objective is to maximize $|M|$.

If we want the worst case with regard to the size of the query range r, selecting any straight line will ensure that we will have as many maximum quadrants as the size of the range. These are the temporal and spatial costs, which are $O(r \log_k n)$ and $O(r)$, respectively.

Since our query does not usually depend on the size of the range, we try to maximize cost with regard to the matrix size.

We try to obtain the highest number of maximum quadrants. Starting at the root and if we want to descend to the highest number of quadrants (k^2 at this point), we must have a minimum query range of 2×2 and a maximum of $n - 1 \times n - 1$, both centered. At the next level, we have $4 \cdot (k - 1)$ quadrants to descend (considering that the four quadrants in the center would be maximum quadrants). This query goes from $\frac{n}{2} + 1 \times \frac{n}{2} + 1$ to $n - 1 \times n - 1$, both centered. Continuing with the previous logic, we can obtain the range of a larger query to be $n - 1 \times n - 1$, as shown in the example of Fig. 3.

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	1	0
1	0	0	0	0	1	0	0	0
2	0	1	1	0	0	0	1	0
3	0	0	1	0	0	1	0	1
4	0	0	0	0	0	1	0	1
5	1	0	1	0	1	1	1	1
6	0	0	0	1	0	0	0	0
7	0	1	0	0	0	0	0	0

FIGURE 3. Specific example of worst case for *Compact Count*.

Lemma 4: The number of maximum quadrants for the worst case is $O(\frac{n}{k})$.

Proof:

Based on the previous explanation, the number of quadrants that are added per level is $q_i = (q_{i-1} - 4) \cdot k + 4 \cdot (2k + 1)$. This occurs because the quadrants in the center of level i are always maximum quadrants; therefore, only the peripheral quadrants are added in the level. For each quadrant of the previous level that is not found in the corner (i.e., $q_{i-1} - 4$), k quadrants are added in this level. For each corner quadrant (exactly four), $2(k + 1)$ are added. The base case of this recurrence equation is $q_2 = k^2$ because no quadrants are ever in the $n - 1 \times n - 1$ range at levels 0 and 1.

The recursive equation is solved by:

$$\sum_{i=2}^{\log_k n} q_i = \sum_{i=2}^{\log_k n} k^i + (4k + 4) \times \sum_{i=0}^{\log_k n - 3} (\log_k n - 2 - i) k^i = O\left(\frac{n}{k}\right). \quad (10)$$

As an alternative solution, the approximation $q_i = O(k^i)$ can be considered if the recurrence is solved by changing the variable and the master theorem (theorem for solving recurrence equations), which leads to the same result as expressed by:

$$\sum_{i=2}^{\log_k n} q_i = \sum_{i=2}^{\log_k n} O(k^i) = O\left(\frac{n}{k}\right). \quad (11)$$

■

The cost of both temporal and spatial *Compact Count* is therefore $O\left(\frac{n}{k} \log_k n\right)$ and $O\left(\frac{n}{k}\right)$, respectively.

E. EXPECTED CASE

To study the expected case, we can perform an analysis similar to the previous worst-case scenario. Assuming that the query range is squared, simply stated as $r = x \times x$, this implies that the largest quadrant that fits in that range has a height of $\log_k n - \lfloor \log_k x \rfloor$. Likewise, we will use the variable $0 < \delta \leq 1$, which represents the density (i.e., $\frac{m}{n^2}$) of the k^2-tree to assume how many quadrants can be empty per level. Therefore, $q_i = \delta \cdot ((q_{i-1} - 4) \cdot k + 4 \cdot (2k + 1))$. If we solve the recurrence equation by changing the variable and master theorem, we will obtain the following three results.

- If $\delta \cdot k > 1$, then $q_i = O((\delta \cdot k)^i)$.
- If $\delta \cdot k = 1$, then $q_i = O(i)$.
- If $\delta \cdot k < 1$, then $q_i = O(1)$.

Therefore, the number of maximum quadrants, depending on δ, is

$$\sum_{i=2}^{\lceil \log_k x \rceil} O(q_i) = \begin{cases} O(x \cdot \delta^{\log_k x} - \delta \cdot k) & \text{if} \quad \delta \cdot k > 1, \\ O(\log_k^2 x) & \text{if} \quad \delta \cdot k = 1, \\ O(\log_k x) & \text{if} \quad \delta \cdot k < 1. \end{cases} \quad (12)$$

The temporal and spatial cost of *Compact Count* increases:

- If $\delta \cdot k > 1$, then the temporal cost is $O((x \cdot \delta^{\log_k x} - \delta \cdot k) \cdot \log_k n)$ and the spatial cost is $O(x \cdot \delta^{\log_k x} - \delta \cdot k)$.

- If $\delta \cdot k = 1$, then the temporal cost is $O(\log_k^2\ x \cdot \log_k n)$ and the spatial cost is $O(\log_k^2\ x)$.
- If $\delta \cdot k < 1$, then the temporal cost is $O(\log_k x \cdot \log_k n)$ and the spatial cost is $O(log_k x)$.

V. EXPERIMENTAL RESULTS

In this section, we present and discuss the results of a series of experiments that show the performance of our algorithm against the solution based on *Range*.

As stated in section II, *Range* query is the standard way that k^2-tree manages counting the relations represented in it. Also, as our objective is not to modify the structure, but to enhance the capacity of k^2-tree, we exclude any modification of the structure in this section, as k^2-treap [16]. In particular, the study of modifications of base structures for aggregate functions (as in [16] and [21]), use traversing the structure as a baseline to its studies.

Both runtime and additional memory (excluding storage of the k^2-tree structure) used by the algorithms for each given data set were measured in the experiments. Synthetic and real data sets were used. Relations were represented by the ordered pair (x, y) with $0 \leq x, y \leq n$ within the $n \times n$ matrix. The algorithm was implemented in the C and C++ language. We used the implementation of the k^2-tree and *Range* available in the library indicated in [20].

Data structure k^2-tree performs well (in terms of space and execution time), when the represented matrix has a very low amount of 1s (sparsed matrix) and/or the relations represented are clustered [1]. In order to study the effects of density and distribution of relations (amount $1s/n^2$) we evaluate sets of synthetic data with different distributions (uniform and clustered) and considering different densities.

We also evaluate our algorithm in real case scenarios. In order to do that, we use real data sets (*snapshots*) from World Wide Web graph from `.uk`, obtained between June 2006 and May 2007 [15]. Eleven indexed sets in a k^2-tree were used on a $1,000,000 \times 1,000,000$ matrix with a mean of 2,240,878 relations (1s). These data sets have been used in different researches as [15] and [1].

The experiment was performed on a server with four Intel(R) Xeon(R) CPU E3-1225 3.30GHz processors with 8192 KB of cache memory. The `clock()` function from the `times.h` library implemented in C was used to measure time. The shell script command `/usr/bin/time` was used to measure memory, whose `-f` option allows obtaining the memory peak used by each process. The resources used during reading/writing of the data were not considered in the measurement.

Runtime and memory usage presented in the following graphs and tables are the average of ten measurements of execution for each data set evaluated. In the case of configurations over synthetic data sets (distribution, amount, range values), we randomly generated ten data sets with different points.

A. RUNTIME

1) NUMBER OF RELATIONS (1s)

The effect of the number of relations (1s) over the algorithms was measured in these experiments. In this scenario, the matrix size was set at $65,536 \times 65,536$ and the query range r at $[1, 65535] \times [1, 65535]$. The size of r represents the worst scenario for *Compact Count*.

Table 2 and Fig. 4 show the runtime of the algorithms for each set and distribution. According to the results in Table 2, *Compact Count* performs better than *Range* in both distributions. More specifically, *Compact Count* is 96 times faster on the average than *Range* (minimum 19, maximum 213) for uniform data sets and 195 times faster on the average (minimum 42, maximum 399) for sets with clustered distribution. Moreover, both algorithms exhibit better behavior in clustered versus uniformly distributed sets. In the case of *Compact Count*, it was three times faster for clustered sets than uniformly distributed sets (see Fig. 4).

TABLE 2. Response time (ms) for number of relations (1s) in *r*.

Number of 1s	Uniform		Clustered	
	Range	Compact Count	Range	Compact Count
100,000	68.02	3.49	63.05	1.48
500,000	287.18	7.12	257.07	2.79
1,000,000	527.94	9.52	472.01	4.90
5,000,000	2,116.41	19.26	1,828.30	8.10
10,000,000	3,763.18	26.85	3,096.41	9.80
50,000,000	13,460.51	63.04	11,107.70	27.85

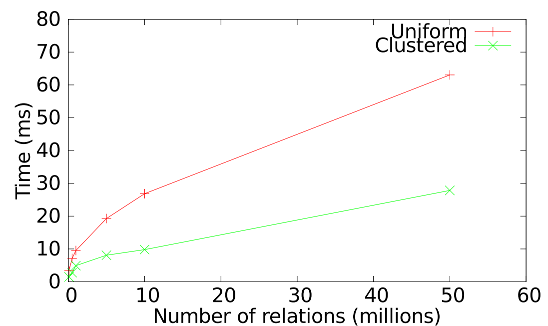

FIGURE 4. Response time (ms) for number of relations (1s) for each distribution in *Compact Count*

Data in Table 2 demonstrates the effects of the number of relations (1s) in the performance of both algorithms. The number of 1s has a greater negative effect on *Range* because of the increased number of branches of the k^2-tree it has to explore to count the 1s in the leaves. In contrast, *Compact Count* must descend the k^2-tree twice for each maximum quadrant, regardless of the number of 1s it contains. *Compact Count* worsens only as the number of non-empty quadrants increases. The increase in non-empty quadrants is slower in the clustered than in the uniform distribution, which leads to a much lower response time for *Compact Count* in the clustered distribution.

2) SET DENSITY

The previous group of experiments does not clearly show the effect of density on the performance of the algorithms. It precisely focuses on evaluating the performance of the algorithms subjected to different densities of the sets of relations. Therefore, matrix size was set at $22,360 \times 22,360$ and range r at $[1, 22359] \times [1, 22359]$, which represents the worst case of *Compact Count*.

Table 3 and Fig. 5 show the runtime of both algorithms for each studied distribution. Based on the data in Table 3, it can be stated that *Compact Count* was 90 and 178 times faster on the average than *Range* for sets with uniform and clustered distribution, respectively; these averages are very similar to those reported in Table 2 for the same distributions. According to Fig. 5, *Compact Count* benefits more from clustering than *Range*. The effect of density on the performance of the algorithms is explained by the same phenomena described in the previous group of experiments (see Section V-A0.a).

TABLE 3. Response time for density (ms).

Density	Uniform		Clustered	
	Range	Compact Count	Range	Compact Count
1	1,403.85	24.84	1,094.94	7.45
5	4,365.52	45.46	3,440.66	18.51
10	6,684.40	57.45	4,357.11	22.24

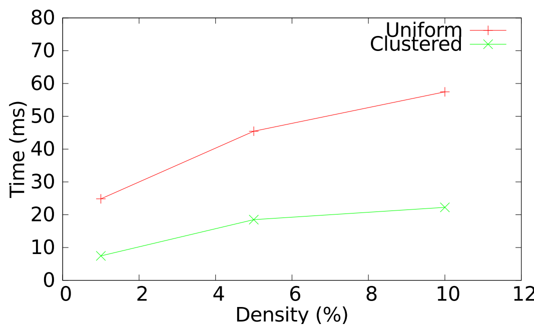

FIGURE 5. Response time for density for each distribution in *Compact Count*.

3) MATRIX SIZE

The aim of this group of experiments is to show the effect of matrix size on the runtime of the algorithms. This involved matrix sizes of $8,192 \times 8,192$, $16,384 \times 16,384$, $32,718 \times 32,718$, and $65,536 \times 65,536$ (see Table 4), sets of 10 million relations (1s), and a query range r $[1, 8191] \times [1, 8191]$.

Table 4 shows that response time usually decreases for both algorithms as matrix size increases. However, *Compact Count* was 48 and 78 times faster on the average than *Range* for the uniform and clustered distribution, respectively. Fig. 6 illustrates that *Compact Count* is approximately two times faster in sets with clustered data than sets with uniform distribution. Density decreases as matrix size increases, maintaining the number of relations constant; this increases the probability of empty quadrants (without 1s), which leverages both algorithms to their advantage. Results in Table 4 and Fig. 6 are consistent with those in Table 3 and Fig. 5, respectively.

TABLE 4. Response time for matrix size (ms).

Matrix size	Uniform		Clustered	
	Range	Compact Count	Range	Compact Count
8,192	252.95	5.20	199.06	2.27
16,384	483.18	7.30	410.23	3.38
32,718	179.09	4.60	87.01	1.50
65,536	60.89	2.81	34.16	1.03

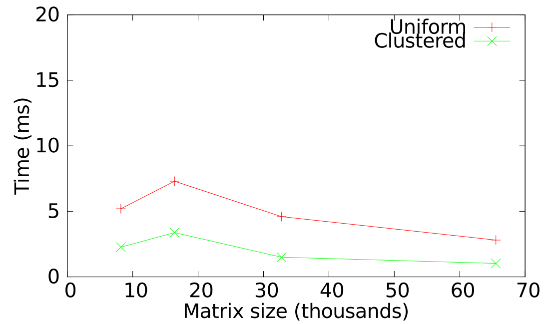

FIGURE 6. Range size response time for each matrix size in *Compact Count*.

4) VARIATION IN SIZE OF QUERY RANGE *r*

The purpose of this group of experiments is to show the effect of the query range size on the runtime of the algorithms. Both the matrix size ($65,536 \times 65,536$) and the sets (10 million 1s) were constant. The size of r was established as a percentage between the size of r and matrix size. The studied ranges were 0.01%, 0.1%, 1.0% 10%,25%, 50%, 99%, and 100%. Squared ranges whose centroids coincide with the centroid of the matrix were considered.

TABLE 5. Response time for range size (ms).

Range coverage (%)	Uniform		Clustered	
	Range	Compact Count	Range	Compact Count
0.01	0.46	0.24	0.33	0.17
0.10	3.85	0.52	3.19	0.41
1.00	37.94	2.60	31.66	2.17
10.00	377.72	13.55	333.51	10.34
25.00	942.63	3.65	877.67	3.51
50.00	1,884.15	19.12	1,825.59	13.52
99.00	3,731.69	50.50	3,215.90	18.71
100.00	3,791.02	0.01	3,239.79	0.01

Table 5 shows that *Compact Count* is far superior to *Range*. It should be noted that as the size of range r increases, the advantage of *Compact Count* is emphasized. Excluding the case in which r is 100% of the matrix size (best case for *Compact Count*), *Compact Count* was between 2 and 258 times faster than *Range*. Even in the worst case for *Compact Count* (99%), it was 73 times faster than *Range*. In the 100% case, *Compact Count* was 380,000 (uniform

distribution) and 324,000 (clustered distribution) times faster than *Range*.

The differences are explained by the fact that as the size of r becomes larger, the number of maximum quadrants in the first levels of the k^2-tree increases and *Compact Count* must perform only two descents per maximum quadrant to count the number of 1s. In contrast, *Range* must perform a descent for each 1 within r, and the number of descents increases as the size of r increases.

Fig. 7 illustrates once again that *Compact Count* is more efficient for clustered data, and the difference significantly increases as the range approximates matrix size. When considering a range r with a size of 100%, the response time of *Compact Count* decreases rapidly.

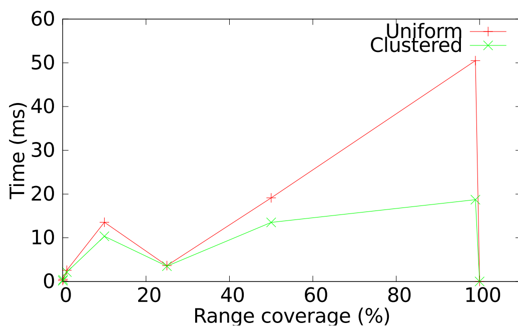

FIGURE 7. Response time for range size for each distribution in *Compact Count*.

5) REAL DATA

The objective of this group of experiments is to evaluate the runtime of the algorithms using real data sets, which are described in Section V. Squared ranges with different sizes and whose centroids coincide with the centroid of the matrix were evaluated (see Table 6).

Table 6 demonstrates the difference in runtime between the algorithms. More specifically, *Compact count* was approximately 900 times faster than *Range*.

TABLE 6. Response time of range size (ms) based on real data.

Range size (%)	Range	Compact Count
0.01%	143.13	0.16
0.10%	148.59	0.14
1.00%	138.26	0.13
10.00%	122.06	0.12
25.00%	114.13	0.13
50.00%	105.21	0.17
75.00%	119.91	0.16
99.00%	128.77	0.15
99.90%	140.97	0.18
99.99%	157.52	0.19
100.00%	164.26	0.15

B. ADDITIONAL MEMORY USED BY THE ALGORITHMS

Additional memory required by the algorithms was also measured in all the experiments (excluding storage occupied by the k^2-tree). Given synthetic data in both the uniform and clustered distribution, additional memory in all the experiments was maintained between 20 KB and 120 KB for both algorithms. In most cases, *Compact Count* required more memory than *Range*, with a maximum of 60 KB additional memory required by *Compact Count*. The experiments also showed the scalability of both algorithms in additional storage against all the studied variables.

In the real data scenario, additional storage was between 10 KB and 40 KB for both algorithms. *Compact Count* reached a maximum of 40 KB and *Range* a maximum of 25 KB.

VI. CONCLUSION

The algorithm called *Compact Count* is proposed in this article to count the number of relations (number of 1s) found in a query range r over a binary relation represented in the k^2-tree compact data structure. The algorithm benefits from the widthwise path performed from the k^2-tree to generate bitmaps T and L. The runtime of the algorithm is limited by $O(\log_k r \log_k n)$ and additional storage $O(\log_k r)$ with r as the query size and n and k as the k^2-tree parameters.

Through a series of experiments, we evaluated the performance of *Compact Count* and compared it with the *Range* algorithm, which is an adaptation of the algorithm that recovers and counts all the relations (1s) in range r. When evaluating synthetic data, the experiments show that *Compact Count* was 250 times faster than *Range* depending on the distribution, size of r, set size, density, and matrix size. When evaluating real data, speed increased to approximately 1,000 times for *Compact Count*. The runtime advantages of our algorithm involve a minimal cost for additional storage that varied between 50 KB and 120 KB, which was generally slightly higher than the one required by *Range*. The maximum difference required by *Compact Count* over the additional memory consumed by *Range* was 60 KB.

An alternative to solve the *Count* problem for the k^2-tree is to store aggregate information to the tree, alternative called k^2-treap. This would reduce the query time of $T = T_{getM} + T_{countM}$ to $T = T_{getM}$ (time to obtain maximum quadrants). Saving time depends on the height of the maximum quadrants (the higher they are, the more time saved); however, in absolute terms, this does not imply an asymptotic decrease in query time, which is still $O(|M| \log_k n)$. Furthermore, for a space-efficient implementation of the structure, the number of points should only be stored in the upper quadrants. In an average case, this means that the temporal acceleration does not exceed $O(\log_k n)$. Given that our objective is to extend the capabilities of the k^2-tree, we seek to minimally modify the structure. We therefore consider that our alternative is more suitable for this objective (mainly because the k^2-tree implementations currently in use already have the structure for *rank* and *select* in the L array).

Given the properties of the compact data structures, such as the k^2-tree, these are emerging as very useful tools for storing and processing large volumes of data directly into memory. Our algorithm is an extension of the processing capabilities

of the k^2-tree. It can benefit other algorithms that need to directly process this query over compressed data within this compact data structure.

REFERENCES

[1] C. Q. Fuentes, M. R. Penabad, S. Ladra, and G. G. Retamal, "Compressed data structures for binary relations in practice," *IEEE Access*, vol. 8, pp. 25949–25963, 2020.
[2] R. Moratz, "Representing relative direction as a binary relation of oriented points," in *Proc. Conf. ECAI, 17th Eur. Conf. Artif. Intell.*, Riva Del Garda, Italy: IOS Press, Sep. 2006, pp. 407–411.
[3] B. Awerbuch, B. Patt-Shamir, D. Peleg, and M. Tuttle, "Improved recommendation systems," in *Proc. 16th Annu. ACM-SIAM Symp. Discrete Algorithms*, Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2005, pp. 1174–1183. .
[4] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener, "Graph structure in the Web," *Comput. Netw.*, vol. 33, nos. 1–6, pp. 309–320, Jun. 2000.
[5] N. R. Brisaboa, S. Ladra, and G. Navarro, "k^2-trees for compact Web graph representation," in *Proc. Int. Symp. String Process. Inf. Retr.*, Springer, 2009, pp. 18–30.
[6] P. Boldi and S. Vigna, "The webgraph framework I: Compression techniques," in *Proc. 13th Conf. World Wide Web WWW*, 2004, pp. 595–602.
[7] P. Papadimitriou, A. Dasdan, and H. Garcia-Molina, "Web graph similarity for anomaly detection," *J. Internet Services Appl.*, vol. 1, no. 1, pp. 19–30, May 2010.
[8] M. Jon Kleinberg, R. Kumar, P. Raghavan, S. Rajagopalan, and S. Andrew Tomkins, "The Web as a graph: Measurements, models, and methods," in *Computing and Combinatorics*, T. Asano, H. Imai, D. T. Lee, S.-I. Nakano, and T. Tokuyama, Eds. Berlin, Germany: Springer, 1999, pp. 1–17.
[9] G. S. Brodal and K. G. Larsen, "Optimal planar orthogonal skyline counting queries," in *Proc. Scandin. Workshop Algorithm Theory*. Springer, 2014, pp. 110–121.
[10] R. Grossi, A. Gupta, and J. S. Vitter, "High-order entropy-compressed text indexes," in *Proc. SODA*, 2003.
[11] J. Barbay, F. Claude, and G. Navarro, "Compact binary relation representations with rich functionality," *Inf. Comput.*, vol. 232, pp. 19–37, Nov. 2013.
[12] N. R. Brisaboa, S. Ladra, and G. Navarro, "Compact representation of Web graphs with extended functionality," *Inf. Syst.*, vol. 39, pp. 152–174, Jan. 2014.
[13] G. Navarro, *Compact Data Structures: A Practical Approach*. Cambridge, U.K.: Cambridge Univ. Press, 2016.
[14] N. R. Brisaboa, G. D. Bernardo, G. Gutiérrez, M. R. Luaces, and J. R. Paramá, "Efficiently querying vector and raster data," *Comput. J.*, vol. 60, no. 9, pp. 1395–1413, Sep. 2017.
[15] C. Quijada-Fuentes, M. R. Penabad, S. Ladra, and G. Gutiérrez, "Set operations over compressed binary relations," *Inf. Syst.*, vol. 80, pp. 76–90, Feb. 2019.
[16] N. R. Brisaboa, G. De Bernardo, R. Konow, G. Navarro, and D. Seco, "Aggregated 2D range queries on clustered points," *Inf. Syst.*, vol. 60, pp. 34–49, Aug. 2016.
[17] D. Papadias, Y. Tao, G. Fu, and B. Seeger, "Progressive skyline computation in database systems," *ACM Trans. Database Syst.*, vol. 30, no. 1, pp. 41–82, Mar. 2005.
[18] H. Miller and J. Han, *Geographic Data Mining and Knowledge Discovery*. Boca Raton, FL, USA: CRC Press, 2009.
[19] W. R. Tobler, "A computer movie simulating urban growth in the detroit region," *Econ. Geogr.*, vol. 46, pp. 234–240, Jun. 1970.
[20] S. Ladra, "Algorithms and compressed data structures for information retrieval," Ph.D. dissertation, Coruña Univ., A Coruña, Spain, 2011.
[21] P. Bose, M. He, A. Maheshwari, and P. Morin, "Succinct orthogonal range search structures on a grid with applications to text indexing," in *Proc. Workshop Algorithms Data Struct.*, Springer, 2009, pp. 98–109.

MARTITA MUÑOZ CANDIA received the degree in computer civil engineering and the M.Sc. degree in computer science from the Universidad del Bío-Bío, in 2017 and 2019, respectively. Her research interest includes data structures and algorithms.

GILBERTO GUTIÉRREZ RETAMAL received the M.Sc. and Ph.D. degrees in computer science from the Universidad de Chile, in 1999 and 2007, respectively. He is currently an Associate Professor with the Department of Computer Science and Information Technology, Universidad del Bío-Bío, Chillán, Chile. His research interests include data structures and algorithms, and spatial and spatio-temporal databases.

RODRIGO TORRES-AVILÉS received the Ph.D. degree in applied mathematics from the Universidad de Concepción, in 2016. He is currently an Assistant Professor with the Departamento de Sistemas de Información, Universidad del Bío-Bío, Concepción, Chile. His research interests include data structures and algorithms, automata theory, and symbolic systems.