IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 18 January 2024, accepted 8 February 2024, date of publication 9 February 2024, date of current version 28 February 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3365054

RESEARCH ARTICLE

cKd-tree: A Compact Kd-tree

GILBERTO GUTIÉRREZ[1], **RODRIGO TORRES-AVILÉS**[2], **AND MÓNICA CANIUPÁN**[2]

[1]Departamento de Ciencias de la Computación y Tecnologías de Información, Universidad del Bío-Bío,Chillán 3800708, Chile
[2]Departamento de Sistemas de Información, Universidad del Bío-Bío, Chillán 3800708, Chile

Corresponding author: Mónica Caniupán (mcaniupan@ubiobio.cl)

This work was supported in part by the Agencia Nacional de Investigación y Desarrollo de Chile (ANID) under Grant 1230647, in part by the Algoritmos y Bases de Datos (ALBA) Group under Grant 2130591 GI/VC, and in part by the Research Grant under Project INES I+D 22-14 and Project 2130520 IF/R.

ABSTRACT In the context of Big Data scenarios, the presence of extensive static datasets is not uncommon. To facilitate efficient queries on such datasets, the utilization of multiple indexes, such as the Kd-tree, becomes imperative. The current scale of managed points may, however, exceed the capacity of primary memory, posing a significant challenge. In this article we introduce cKd-tree, a compact data structure designed to represent a Kd-tree efficiently. The structure cKd-tree is essentially an encoding of the spiral code sequence of points within an implicit Kd-tree (iKd-tree) using Directly Addressable Codes (DACs). The unique feature of cKd-tree lies in its ability to perform spiral encoding and decoding of points by relying solely on knowledge of their parent points within the iKd-tree. This inherent property, combined with DACs' direct access capability to sequence elements, enables cKd-tree to traverse and explore the tree while decoding only the nodes relevant to queries. The article details the algorithms necessary for creating and manipulating a cKd-tree, as well as algorithms for evaluating two fundamental queries over points: the *point query* and the *range query*. To assess the performance of cKd-tree, a series of experiments are conducted, comparing it with iKd-tree and k^2-tree data structures. The evaluation metrics include compression efficiency and execution time of queries. cKd-tree achieves a compression ratio comparable to that of k^2-tree, approximately 70%, demonstrating heightened efficiency, particularly in scenarios characterized by sparse data. Additionally, consistent with expectations, k^2-tree exhibits superior performance in querying individual points, whereas cKd-tree outperforms in the context of aggregate data queries, such as range queries.

INDEX TERMS Compression, indices, spatial data, spatial points, spatial queries.

I. INTRODUCTION

Consider a set of points of interest denoted as S, where the requirement involves executing multiple queries over this set. Various multidimensional data structures have been devised for the storage and efficient querying of such points. Noteworthy among these structures are the Kd-tree [1], BSP-tree [2], and Quadtree [3] (for further insights, see [4] and [5]). These structures enable the execution of diverse single and aggregate queries without necessitating a full scan of the entire dataset. However, these structures also provide the flexibility to modify the set S through operations such as insertion or deletion of points, albeit at the cost of requiring additional $O(|S|)$ space for storing pointers.

Many of these structures are designed with the assumption that sets of points are dynamic, meaning they can grow or shrink over time, necessitating support for operations like insertions and deletions. Consequently, these structures are tailored to handle dynamic sets efficiently. In contrast, when dealing with static sets-where the number of points remains constant over time-data structures can focusing on implementing operations that do not alter set size. This results in more cost-effective implementations in terms of storage, as the algorithm proposed in [6] that constructs a static Kd-tree in $O(dn \log n)$ time. This static Kd-tree is implicitly stored in an array, without using pointers, making it occupy less storage while maintaining navigation efficiency comparable to its dynamic counterpart.

Over the past few decades, there has been a pronounced surge in the exploration and development of compact data structures (CDS), drawing significant interest from both

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

academic and industrial sectors. In essence, a compact data structure compresses static data, minimizing storage requirements and facilitating data processing without the need for prior decompression. The latter characteristic empowers the processing of larger datasets directly in main memory. Additionally, the size of the dataset influences the placement of the compact data structure in the memory hierarchy, potentially positioning it at higher levels (closer to the CPU). This strategic placement enhances overall performance, offering a versatile and resource-conscious approach to managing and querying datasets. The utilization of compact data structures presupposes the static nature of the data, or alternatively, a substantially lesser frequency of modifications compared to the frequency of queries.

An illustrative example is found in [7], where the authors introduce the k^2-tree, a CDS originally designed for representing web graphs but applicable to encoding any binary relationship. In information retrieval, foundational CDSs encompass Wavelet trees [8], [9], the Compressed Suffix Array (CSA) [10], [11], and the FM-Index [12]. Furthermore, CDSs have found utility in diverse fields like Geographical Information Systems, where they optimize query processing [13], [14], [15]. For an insightful overview of compact data structures, the book [16] stands as an excellent resource, providing a comprehensive review of the advancements and applications in this evolving domain.

The data structures employ a point encoding mechanism to store information efficiently, reducing the amount of data that needs to be stored. One such encoding method is the spiral code, as utilized in [17]. The spiral code involves storing the distance from each point to its parent point, resulting in a more concise representation of information. This approach minimizes the amount of registered data, optimizing storage efficiency.

In this work, we introduce a compact version of the Kd-tree designed for storing a set of static points $S \subseteq \mathbb{N}^2$, named the cKd-tree. We focused our attention on the Kd-tree, given its prominence as one of the most widely employed structures for indexing spatial and geometric data [4]. The cKd-tree we propose utilizes a spiral encoding scheme to encode the points within the implicit Kd-tree, resulting in a sequence C of integers. This sequence is then represented using Directly Addressable Codes (DACs), a method that encodes integer numbers and facilitates direct access [18]. The resulting cKd-tree achieves a highly efficient and minimalistic representation of the underlying static point set.

The cKd-tree demonstrates a notable reduction (approximately 70%) in storage requirements, preserving the capability to execute standard Kd-tree queries. However, it incurs an additional querying cost of $O(\log_2 m)$ per node, where m signifies the maximum value within the sequence C. This supplementary cost becomes negligible when dealing with a vast number of points, given that iKd-tree, due to its inability to fit into main memory, necessitates queries involving secondary memory. In contrast, our proposal exhibits a significantly higher threshold for fitting within main memory. Notably, this compact data structure introduces the advantage of requiring decoding solely for the points along the navigation path of the tree during query operations.

We present the algorithms for constructing and querying the structure, accompanied by a theoretical and experimental analysis in comparison to the baseline k^2-tree. As anticipated, the comparison with k^2-tree reveals that for individual data queries, such as determining if a point belongs to S, our proposal is comparatively slower (similar to the original iKd-tree). However, when querying aggregate data, as in the case of a range query, our proposal demonstrates superior efficiency.

The subsequent sections of this article are structured as follows: Section II delves into related work, offering a contextual overview. In Section III, the foundational data structures and algorithms that underpin our proposed structure are expounded upon. Section IV provides an exploration of the cKd-tree compact data structure, accompanied by detailed discussions on query processing algorithms (Section IV-B) and a comprehensive analysis of complexity (Section IV-C). Moving forward, Section V presents and discusses the experimental results. Finally, Section VI presents the conclusions drawn from this work and outlines directions for future research.

II. RELATED WORK

Various dynamic data structures are available for storing points in both primary and secondary memory. One of the earliest and widely adopted structures is the Kd-tree [1]. A Kd-tree is a hierarchical structure, specifically a binary tree, where the space or subspace undergoes recursive subdivision through iso-oriented hyperplanes of $d - 1$ dimensions [4]. Notably, the efficacy of a Kd-tree is contingent upon the order in which objects are inserted. However, despite this sensitivity, insertion, deletion, and search operations, on average, exhibit temporal bounds of $O(\log_2 n)$, with n denoting the size of the point sets.

The BSP-tree, very similar to the Kd-tree, is a multidimensional data structure that shares the concept of recursively partitioning space using $d - 1$ dimensional hyperplanes. Notably, these hyperplanes in the BSP-tree need not be iso-oriented. In contrast to the Kd-tree, where partitions alternate between different dimensions, the BSP-tree adapts its partitions based on the distribution of objects within the space or subspace.

Another notable data structure is the Quad-tree [3], renowned for its various variants. This structure, like the Kd-tree, employs recursive division of space or subspace through iso-oriented hyperplanes. However, a key distinction lies in the fact that each internal node of the Quad-tree typically has 2^d children. Unlike Kd-trees, these structures are predominantly designed for dynamic object sets and are commonly implemented using pointers.

In recent developments, these data structures have undergone adaptations to accommodate static sets of multidimensional objects. Notably, in [6], an algorithm was introduced for constructing a static Kd-tree tailored for a set of points S with a height of $O(\log_2 n)$, where $n = |S|$. This implementation eliminates the need for pointers, directly storing points in an array. The resulting structure is referred to as iKd-tree [19]. To achieve this efficiency, the algorithm initially sorts the set along each dimension, incurring a total cost of $O(d \cdot n \log_2 n)$. Subsequently, in a second step with a time complexity of $O(n)$, the algorithm selects the median of the array as the root node of the iKd-tree, proceeding recursively with each sub-array and alternating the coordinate for partitioning [19].

Moreover, compact data structures offer an alternative for representing points, exemplified by the k^2-tree [7]. In this structure, points are derived from an adjacency matrix A, where a point $p(x, y)$ in the set is symbolized by a stored value of 1 in the cell $A[x, y]$. Essentially, the set of points forms a subset of the Cartesian product across d dimensions. The k^2-tree is constructed recursively, starting with a partition of k^2 sub-matrices within matrix A, leading to the creation of the root node of the k^2-tree with k^2 children. Sub-matrices lacking points (entirely containing zeros) remain unpartitioned, with a corresponding zero bit recorded in those nodes. Conversely, sub-matrices containing at least one cell with the value 1 undergo recursive subdivision, generating a new internal node where a bit is set to 1 and stored. This process continues until reaching cells of size 1×1. The versatility of this compact structure extends across diverse application domains, including graph representations [7], raster data [20], and points in $\mathbb{N}^2$ [14]. Noteworthy is the k^2-tree's ability to store static sets of points without relying on pointers, making it a foundational structure for comparative analysis in our study.

III. BACKGROUND

This section provides an overview of the foundational data structures that serve as the basis for the proposed cKd-tree compact data structure.

A. KD-TREE DIMENSIONAL DATA STRUCTURE

A Kd-tree, short for k-dimensional tree, is a hierarchical structure (binary tree), designed for recursive division of multi-dimensional space [1]. Within this structure, each node contains data representing a d-dimensional point in the space.

A non-leaf node within a d-dimensional Kd-tree strategically partitions the space, creating two distinct halves or half-spaces. Points located to the left of this partition are correspondingly represented by the left sub-tree of the node, while those on the right find representation in the right sub-tree. The root of the Kd-tree aligns with an x-oriented plane, while its immediate children align with y-oriented planes and so on. At height d, the internal nodes revert to x-oriented planes, creating a pattern that continues throughout the tree. For a more concrete illustration, consider Example 1, that illustrate the construction process of a Kd-tree for a 2D space (where $d = 2$).

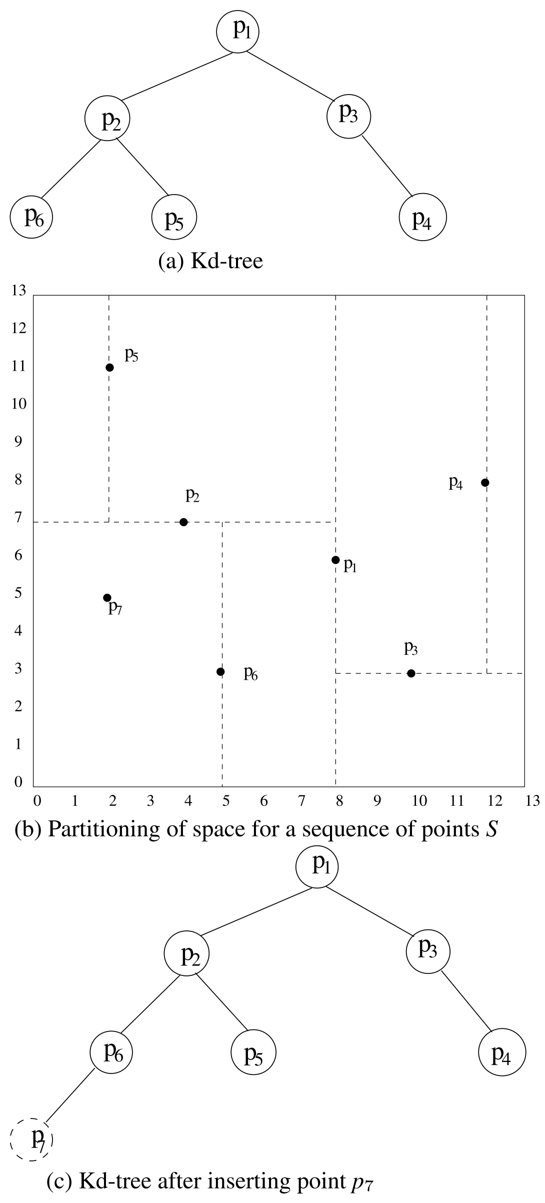

p1
p2
p3
p6
p5
p4
(a) Kd-tree
0 1 2 3 4 5 6 7 8 9 10 11 12 13
p5
p2
p4
p1
p7
p6
p3
(b) Partitioning of space for a sequence of points S
p1
p2
p3
p6
p5
p4
p7
(c) Kd-tree after inserting point p7

FIGURE 1. **A Kd-tree and its spatial partitions.**

Example 1: Consider the following sequence of points $S = \langle p_1(8, 6), p_2(4, 7), p_3(10, 3), p_5(2, 11), p_6(5, 3), p_4(12, 8)\rangle$ *with the x coordinate serving as the basis for space partitioning at level 0, which corresponds to the root of the tree. The construction of the Kd-tree unfolds as follows:*

- *Initially, the point* $p_1(8, 6)$ *is inserted as the root of the tree, prompting a partition along the x axis at* $x = 8$.
- *Subsequently, when point* $p_2(4, 7)$ *is inserted, its x coordinate being smaller than that of* $p_1(8, 6)$ *places it in the left subspace of the partition created by* $p_1(8, 6)$*. Another consequence of inserting* $p_2(4, 7)$ *is the partitioning of this subspace based on its y coordinate. Consequently, subsequent comparisons involving* $p_2(4, 7)$ *will prioritize the evaluation of the y coordinate.*

- *Point $p_3(10, 3)$ is introduced into the structure, positioned to the right of $p_1(8, 6)$ owing to its higher x coordinate value. The insertion of $p_3(10, 3)$ further partitions the subspace to the right of $p_1(8, 6)$ based on the y coordinate.*
- *Subsequently, when inserting point $p_5(2, 11)$ to the right of point $p_2(4, 7)$, the decision is made based on its x coordinate being smaller than that of $p_1(8, 6)$ and its y coordinate being greater than that of $p_2(4, 7)$.*
- *Similarly, point $p_6(5, 3)$ is inserted to the left of $p_2(4, 7)$, while point $p_4(12, 8)$ is positioned to the right of $p_3(10, 3)$.*

Figure 1 depicts the Kd-tree (shown in Figure 1a) corresponding to the given list of points in S, considering the partitioned space illustrated in Figure 1b.

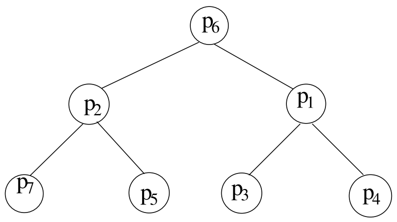
The algorithm to search a point is very similar to the insertion algorithm. For instance, to determine whether point $p_7(2, 5)$ is within the Kd-tree (refer to Figure 1b), the algorithm initiates its search from the root, traversing to the leaves while inspecting if p_7 corresponds to a point stored in each visited node. In cases where p_7 does not match the point at the current node, a decision is made to continue the search based on either the x or y coordinate. Ultimately, upon reaching a leaf node without finding p_7, it is concluded that $p_7(2, 5)$ is not part of the set. Figure 1c visually demonstrates the insertion of point $p_7(2, 5)$, placed to the left of $p_6(5, 3)$. This insertion prompts the partitioning of the subspace based on the y coordinate of p_7.

According to [1], the fundamental operations of the Kd-tree, on average, exhibit a temporal bound of $O(\log_2 n)$, where n represents the number of points in the sets. The conventional implementation of a Kd-tree typically employs pointers, facilitating the representation of dynamic sets of points and enabling mixing insertion, deletion, and search operations.

B. IKD-TREE: IMPLICIT KD-TREE

We refer to a balanced Kd-tree represented in an array as iKd-tree. In this implementation, pointers are unnecessary, as highlighted by [6]. The construction process of iKd-tree is described in [6].

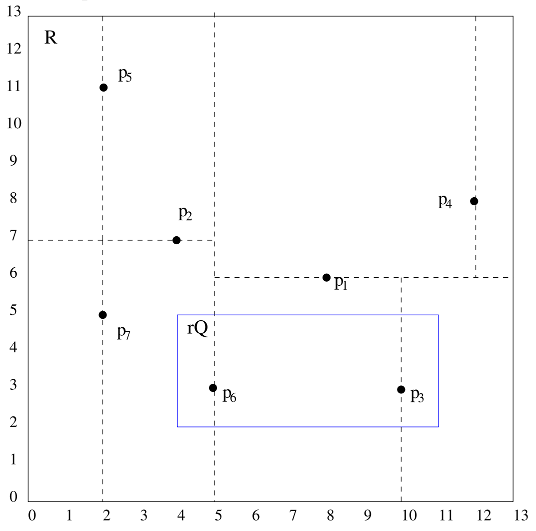
As an illustration, Figure 2 represents an iKd-tree constructed from the set of points $S = \langle p_1(8, 6), p_2(4, 7), p_3(10, 3), p_5(2, 11), p_6(5, 3), p_4(12, 8), p_7(2, 5)\rangle$ in Example 1. The creation algorithm for an iKd-tree first sorts the set of points in ascending order considering both coordinates. This sorting process is performed only once for each coordinate, yielding two arrays: $OX = [p_7(2, 5), p_5(2, 11), p_2(4, 7), p_1(8, 6), p_3(10, 3), p_4(12, 8)]$, containing the points sorted by x coordinates, and $OY = [p_6(5, 3), p_3(10, 3), p_7(2, 5), p_1(8, 6), p_2(4, 7), p_4(12, 8), p_5(2, 11)]$, containing the points ordered by y coordinate. In the subsequent step, with x as the partition coordinate for the first level, the array OX is split into two subsets based on the median point of OX. Following this, the array OY is divided, taking into account the median of OX and comparing it with the x coordinate. Subsequently,



(a) iKd-tree for a set of points S

i	1	2	3	4	5	6	7
Point	p_7	p_2	p_5	p_6	p_3	p_1	p_4
Q	(2,5)	(4,7)	(2,11)	(5,3)	(10,3)	(8,6)	(12,8)
C	17	76	75	0	32	25	51

(b) Array Q storing the iKd-tree in Figure 2a. Array C storing the spiral codes of points in Q (see Section III)



(c) Partition generated by the iKd-tree for the space R containing the points from S, and the query rectangle rQ defined by the coordinates (4,2) and (11,5)

FIGURE 2. iKd-tree representation for the set of points *S* in Example 1.

the two subarrays are partitioned, this time considering the Y coordinate of the median points. This alternating procedure of partitioning coordinates continues until each subset contains a single point. The resulting arrangement is stored in array Q (Figure 2b).

The construction cost of an iKd-tree is $O(d \cdot n \cdot \log_2 n)$. The algorithm ensures the creation of a balanced binary tree, with a tree depth of $\log_2(n)$, as depicted in Figure 2a. Navigating or exploring the iKd-tree is straightforward. Generally, the position of the root node within the subtree, defined by the initial positions li and final positions ls of array Q, (refer to Figure 2b) is situated at position $\left\lfloor \frac{li+ls}{2} \right\rfloor$.

C. SPIRAL CODIFICATION OF POINTS

The spiral encoding method involves assigning a positive integer to a point p in $\mathbb{N}^2$ based on another point q. This encoding approach was applied in [17] for encoding trajectories of moving objects.

Formally, considering points $p(x, y)$ and $q(x, y)$ from the set $S \subseteq \mathbb{N}^2$ with dimensions $|X|$ and $|Y|$, the spiral encoding of q with respect to p is denoted as a function $sCode_p : \mathbb{N}^2 \rightarrow$

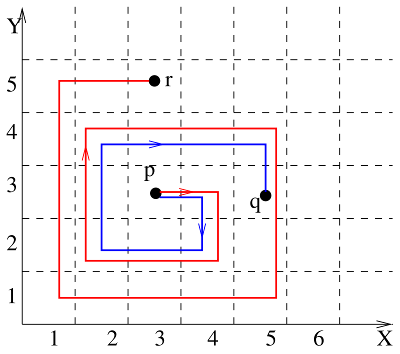


FIGURE 3. Spiral encoding of a set of points in $\mathbb{N}^2$.

$0, \ldots, |X| \cdot |Y|$. In this representation, $sCode_p(q)$ represents the distance, measured in the number of cells, from the cell of p to the cell of q following a spiral path with the origin at p (cell 0). For clarification, Figure 3 illustrates the spiral paths used to encode points q and r starting from the reference point p. In this example, $sCode_p(q)$ is equal to 10, and $sCode_p(r)$ is equal to 22.

On the other hand, the function $sDecode_p(q) : 0, \ldots, |X| \cdot |Y| \rightarrow \mathbb{N}^2$ facilitates retrieving the coordinates of a point q from its spiral code with respect to the point p.

D. DACS

The acronym DACs stands for Directly Addressable Codes [18], which represent a variable-length encoding of sequences of non-negative integers, typically arrays.

Consider an array $X = x_0, \ldots, x_{n-1}$ of non-negative integers, and let b be the block size. Each integer in the array X is encoded using *Vbyte coding*. This method divides the binary representation of x_i (where the size is at least $\log_2 x_i + 1$) into blocks of b bits. Each block is then stored in a chunk of $b+1$ bits. The most significant bit of the chunk is set to 0 when the chunk holds the most significant bits of x_i, and 1 otherwise.

After obtaining the Vbyte code for each $x_i \in X$, the chunks are grouped in order. Subsequently, the Vbyte codes of all x_i in the array are arranged into streams $C = C_0, \ldots, C_{L-1}$, where L is the maximum value of m such that $m = \lceil \frac{\lfloor \log_2 x_i + 1 \rfloor}{b} \rceil$, for all $x_i \in X$.

Each stream C_j consists of two parts. The most significant bit of each C_j is concatenated into the bitmap B_j, while the least significant b bits are stored contiguously in an array A_j.

Example 2: Consider $X = \{25, 2, 70, 10\}$ *and* $b = 3$. *The Vbyte codes are* 0011 1001, 0010, 0001 1000 1110 *and* 0001 1010, *respectively. Then*:

- C_1 *consists in array* $A_1 = \{001, 010, 110, 010\}$, *and the bitmap* $B_1 = \{1, 0, 1, 1\}$.
- C_2 *consists in array* $A_2 = \{011, 000, 001\}$, *and the bitmap* $B_2 = \{0, 1, 0\}$.
- C_3 *consists in array* $A_3 = \{001\}$, *and the bitmap* $B_3 = \{0\}$.

Bitmaps B_j are enhanced with an additional structure of size $o(|B_j|)$ to facilitate constant-time *rank* operations [21]. The *rank* operation on a bitmap B provides the count of 1s (or 0s) up to a given position i. For example, given $B = 0010110101011$, the operation $rank_1(B, 6) = 3$ indicates that there are three occurrences of 1s in B before position 6 (inclusive). Since we solely use *rank* for counting 1s, the sub-index is omitted in this document (*i.e., rank*$_1()$ is denoted as *rank*()).

The overall structure encompasses the B_j bitmaps, their corresponding rank structures, the A_j arrays of bitmaps, the chunk size b, and pointers to these bitmaps and arrays.

Example 3: Consider the data in Example 2. To locate the second element of the array, we begin with $C_1[2] = 1 : 110$. *Since* $B_1[2] = 1$, *we set* $i_2 = rank(B_1, 2) = 1$. *Continuing with* $C_2[1] = 1 : 000$, *as* $B_2[1] = 1$, *we then have* $i_3 = rank(B_2, 1) = 0$. *Finally, concluding with* $C_3[0] = 0 : 001$, *we obtain* $001000110 = 70$.

The worst-case scenario for a search is at most $O\left(\frac{\log M}{b}\right)$ accesses, where $M = \max X$ [18].

Although we have considered a fixed block size b for DACs, it is possible to choose a different block size at each level l (b_l). This flexibility can be advantageous to achieve specific goals, such as optimizing compression. Additionally, modifications can be made to either consider fewer levels or strike a balance between achieving good access times and optimal compression, potentially resulting in a larger last level. In this proposal, distinct block sizes for b are strategically employed across various levels, aiming to optimize the compression outcome.

IV. OUR PROPOSAL

In this section, we introduce cKd-tree, a compact data structure designed for representing points, accompanied by algorithms for efficient querying. Additionally, we provide a comprehensive analysis of the complexity in terms of both storage and query execution time.

cKd-tree is built upon the implicit version of a Kd-tree and leverages spiral encoding to represent a set of points in a multidimensional (d-dimensional) space. Moreover, cKd-tree employs DACs encoding for a sequence of integers (refer to Section III for details). This dual encoding approach enables the structure to store information more efficiently. Specifically, the distance from each node to its parent is registered using spiral coding, and subsequently, the entire spiral codification array-a positive sequence of integers-is encoded using DACs, allowing for direct access.

Formally, let $S \subseteq \mathbb{N}^2$ be a set of points. A cKd-tree is represented as a tuple $\langle Seq, p \rangle$, where Seq is a sequence of integers encoded using DACs, representing an iKd-tree, and $p \in S$ denotes the root of the tree. The sequence Seq is derived from the DACs encoding through a spiral codification of the points in the iKd-tree.

A. CONSTRUCTION OF A CKD-TREE

Algorithm 1 outlines the construction process of a cKd-tree for a set of points S. Initially, the function *CreateiKdtree*() generates an iKd-tree, which is then stored in an array Q (refer to Figure 2). Subsequently, the function *CreateCodecKd-tree*() (see Algorithm 2) is employed to produce a sequence of integers (depicted as array C in Figure 2b). The resulting sequence is represented using the DACs encoding and compression method (Algorithm 1, Line 7).

Algorithm 1 Creation of a cKd-tree

```
1: CreatecKdtree(ArrayofPoints $S$)
2: $Q = CreateiKdtree(S)$
3: $mid = \left\lfloor \frac{|Q|}{2} \right\rfloor$
4: $p = Q[mid]$
5: sea $C[1 \ldots |Q|]$ {$C$ store the spiral codes}
6: CreateCodecKd-tree ($Q, p, 1, |Q|, C$)
7: $Seq = CreateDACs(C)$
8: return $\langle Seq, p \rangle$
```

Algorithm 2 Generation of a Spiral Code From Points in a iKd-tree

```
1: CreateCodecKd-tree (ArrayofPoints $Q$, Point $p$, int $li$,
   int $ls$, ArrayofInteger $C$)
2: if $ls \geq li$ then
3:     $mid = \left\lfloor \frac{li+ls}{2} \right\rfloor$
4:     $C[i] = sCode(p, Q[mid])$
5:     CreateCodecKd-tree ($Q, Q[mid], li, mid - 1, C$)
6:     CreateCodecKd-tree ($Q, q[mid], mid + 1, ls, C$)
7: end if
```

B. PROCESSING OF QUERIES OVER A CKD-TREE

In this section, we delve into algorithms addressing two fundamental queries on sets of points: the *point query* and the *range query* for a cKd-tree. The point query determines whether a given point q is part of the set of points represented by the cKd-tree, while the range query retrieves all points stored in the cKd-tree within the bounds of the specified iso-oriented rectangle for the query range.

Algorithm 3 details the computation of the point query for a cKd-tree, which closely mirrors the process for a iKd-tree, with the only exception being Line 4. This discrepancy arises because the point in the internal node is encoded and necessitates recovery. Initially, the algorithm retrieves the spiral code of the current point p from the DACs Seq using the *Access()* function. Following this, it decodes the coordinates of the current point using the *sDecode()* function (refer to Section III-C). It is crucial to note that during the execution of this query, decoding is only required for the points situated on the path defined by the query point q in the cKd-tree.

The *GetCoor()* function extracts the coordinates (x or y) of p and q based on the partition coordinate stored in the variable *coor*. If *coor* = true, the x coordinate serves as the partition coordinate (generating a vertical partition), and y coordinate otherwise (generating a horizontal partition). Analogous to the iKd-tree, cKd-tree achieves a balanced Kd-tree by compromising discrimination ability. There is a possibility of having more than one point with a coordinate identical to the partition coordinate of the median point. In such cases, a search is required in both subspaces (Lines 15-18).

Example 4: Let's consider the verification of whether the point $q(3, 2)$ is present in the cKd-tree generated from the array C in Figure 2b. The initial call to PointQuery() involves $p = p_6(5, 3)$, $q(3, 2)$, $li = 1$, $ls = 7$, coor = true, and ck containing the DACs encoding of the cKd-tree.

The root point p of the cKd-tree (which is part of the cKd-tree) is used as the initial reference point in the spiral decoding, with its encoding being 0 (Array C in Figure 2b). At Line 4, we retrieve the spiral code of p from Seq using the DACs Access() function. Subsequently, with the help of the sDecode() function, we obtain the point $p(5, 3)$ for comparison with $q(3, 2)$. As these points differ and the node of p is not a leaf, we proceed to traverse down the tree.

In this case, the comparison is based on the x coordinates of p and q to decide which subtree to explore further. Another call is made to PointQuery with $p(5, 3)$, $q(3, 2)$, $li = 1$, $ls = 3$, and coor = false. At Line 4 the function Access() returns the spiral code 76. Following this, using the sDecode() function and the point $p(5, 3)$, we obtain the point $p(4, 7)$, which is again compared with q, this time considering the y coordinates of both points.

The PointQuery function is called once more with $p(4, 7)$, $q(3, 2)$, $li = 1$, $ls = 1$, and coor = true. In this instance, Line 4, the point $p(2, 5)$ is recovered. This point is then compared with $q(3, 2)$, taking into account their x coordinates. Since $q(3, 2)$ is expected to be in the subtree to the right of $p(2, 5)$, and $p(2, 5)$ resides in a leaf node, the algorithm concludes that $q(3, 2)$ is not found in the cKd-tree.

Algorithm 3 Compute *PointQuery()* Over a cKd-tree

```
1: PointQuery(cKd-tree $ck$, Point $p$, Point $q$, int $li$, int $ls$,
   bool $coor$)
2: if $li \geq ls$ then
3:     $i = \left\lfloor \frac{(li+ls)}{2} \right\rfloor$
4:     $p = sDecode(Access(ck.Seq, i), p)$
5:     if $p = q$ then
6:         return $true$
7:     end if
8:     $c_p$ = GetCoor($p, coor$)
9:     $c_q$ = GetCoor($q, coor$)
10:    if $c_q < c_p$ then
11:        return PointQuery($ck, p, q, li, i - 1, \neg coor$)
12:    else if $c_q > c_p$ then
13:        return PointQuery($ck, p, q, i + 1, ls, \neg coor$)
14:    else
15:        $r$ = PointQuery($ck, p, q, li, i - 1, \neg coor$)
16:        if $\neg r$ then
17:            return PointQuery($ck, p, q, i + 1, ls, \neg coor$)
18:        end if
19:    end if
20: else
21:    return $false$
22: end if
```

Algorithm 4 details the computation of the range query over a cKd-tree. This algorithm follows the logic employed for computing such queries over a iKd-tree, with the exception of Line 5. The core functionality of the algorithm hinges on understanding the spaces occupied by the set of points (R in the Algorithm) and the range query rQ. Both R and rQ are iso-oriented rectangles and are defined by two points, corresponding to the extreme points of the main or secondary diagonal.

Similar to the preceding algorithm, this one retrieves the coordinates of the points in the cKd-tree (Line 5). Subsequently, it recursively assesses whether the current rectangle, encompassing the current point p, intersects with the rectangle rQ (Line 6). Following this, it determines the position of the current point p within rQ (Line 8).

The algorithm then divides the current rectangle R using the point p and the suggested coordinate from the iKd-tree. It continues the recursive checking process with both resulting rectangles (Line 11). Similar to Algorithm *PointQuery()*, this algorithm selectively decodes points that are part of the traversal process.

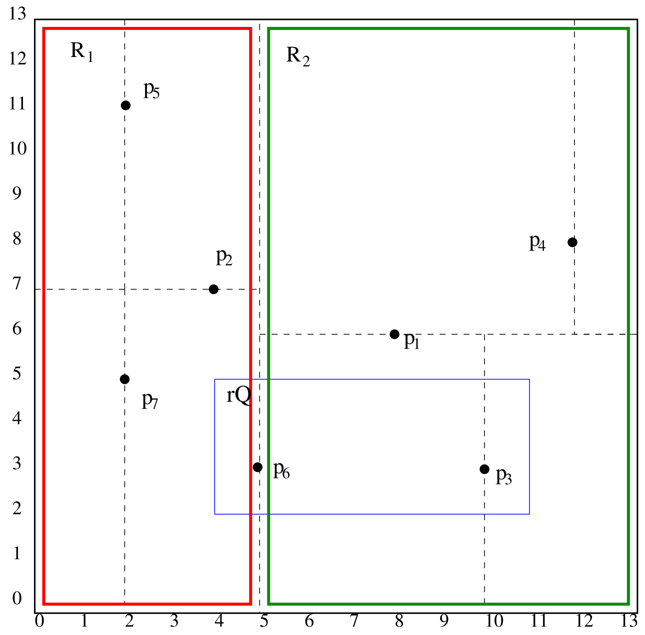


FIGURE 4. Range query.

Example 5: Let's consider the task of identifying points within the range of the rectangle rQ, as illustrated in Figure 2. As in Example 4 the parameters are $p = p_6(5,3)$, $li = 1$, $ls = 7$, $coor = true$, *ck containing the DACs encoding of the cKd-tree, and R being the rectangle, defined by points* $(0,0)$ *and* $(13,13)$, *and that contains the root point p. Also, as in Example 4, the root point p of the cKd-tree is used as the initial reference point in the spiral decoding, with its encoding being* 0, *and then in Line 5 (Algorithm 4) we obtain the point* $p(5,3)$.

Given that the rectangles rQ of the range query and the current rectangle R intersect, it is checked whether $p(5,3)$ *is located within rQ. Given the affirmative,* $p(5,3)$ *is added to the solution* ($sol = \{(5,3)\}$). *In Line 10 (Algorithm 4), using the CreateRectangles() function, two rectangles* R_1 *and* R_2 *are created by dividing R according to point* $p(5,3)$ *and the* $coor = true$ *parameter (which means divided vertically). This is illustrates in Figure 4.*

Subsequently, the algorithm recursively explores the subspaces R_1 *and* R_2, *seeking points that lie within rQ. For* R_1 *(similarly for* R_2*), the call is initiated with the values* $li = 1, ls = 3, p(5,3), coor = false$, *and* $R = R_1$. *Now, in Line 5 (Algorithm 4), the point* $p(4,7)$ *is recovered, but since it is not within rQ, it is not added to the solution. The subspaces associated with the points* $p(2,11)$ *and* $p(2,5)$ *are explored recursively, and they are similarly excluded from the set sol. Upon exploring* R_2, *the point* $p(10,3)$ *is added to the solution. Ultimately, the set* $\{(5,3),(10,3)\}$ *is returned.*

Algorithm 4 Compute *RangeQuery()* Over a cKd-tree

```
1: RangeQuery(cKd-tree $ck$, Point $p$, Rectangle $rQ$,
   Rectangle $R$, int $li$, int $ls$, bool $coor$)
2: $sol = \emptyset$
3: if $li \geq ls$ then
4:   $i = \left\lfloor \frac{(li+ls)}{2} \right\rfloor$
5:   $p = sDecode(Access(ck.Seq, i), p)$
6:   if IntersectRectangles($rQ, R$) then
7:     if Inside($p, rQ$) then
8:       $sol = sol \cup \{p\}$
9:     end if
10:    $\langle R_1, R_2 \rangle$ = CreateRectangles($R, p, coor$)
11:    $sol = sol \cup$ RangeQuery($ck, p, rQ, R_1, li, i - 1, \neg coor$) $\cup$ RangeQuery($ck, p, rQ, R_2, i + 1, ls, \neg coor$)
12:  end if
13: end if
14: return $sol$
```

C. COMPLEXITY ANALYSIS

In this section we present the complexity analysis of the cKd-tree compact data structure. Let $n = |S|$ denotes the number of points on a structure cKd-tree, and let m be the maximum spiral code. As in [18], the time complexity of the *Access()* operation is proven to be asymptotically bounded above by $O(\log m)$.

The *PointQuery()* algorithm (see Algorithm 3) traverses the implicit tree structure, decoding the spiral code at each internal node labeled i with the value $Seq[i]$. Its time complexity is upper bounded by $O(\log n \cdot \log m)$. This analysis assumes that in each recursive call, the values c_p and c_q are unequal, which is the expected scenario (Lines 8 and 9).

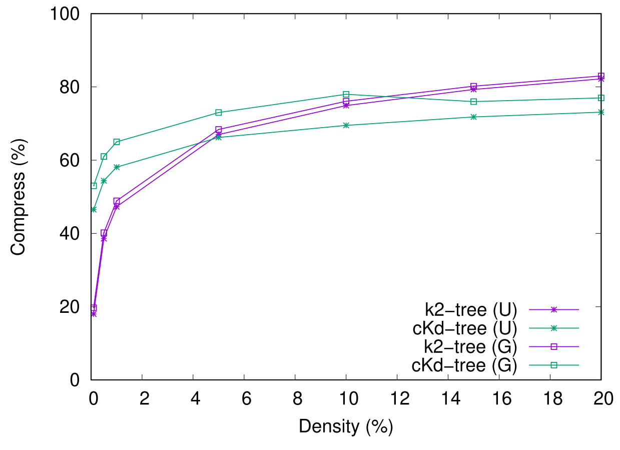
In scenarios where certain recursive calls result in $c_p = c_q$, the time complexity still adheres to the upper bound of $O(\log n \cdot \log m)$. However, it's crucial to acknowledge a potential exception-a pathological case where this upper bound is notably exceeded. This situation arises when we store a straight line and subsequently query for a point along that line. In such pathological cases, a consistent pattern

TABLE 1. Storage (in Mbytes) and compression percentage (%C) for a space of size $16,384 \times 16,384$.

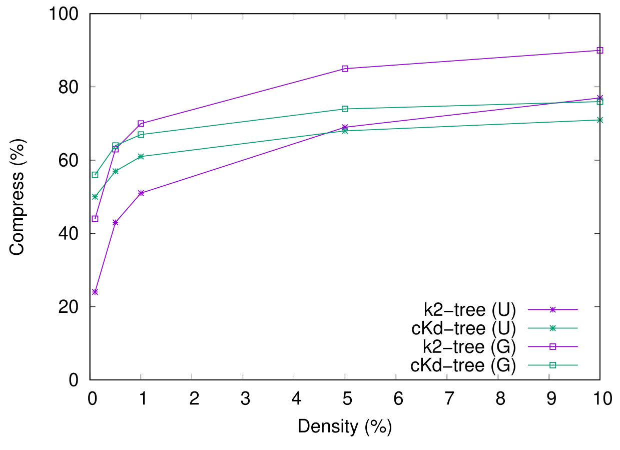
		Uniform				Gaussian			
Density	iKd-tree	k^2-tree		cKd-tree		k^2-tree		cKd-tree	
%	MB	MB	%C	MB	%C	MB	%C	MB	%C
0.1	0.9	0.7	18.0	0.5	46.5	0.5	19.7	0.4	53.0
0.5	4.5	2.9	38.6	2.0	54.4	1.8	40.2	1.7	61.0
1.0	8.0	4.7	47.3	3.7	58.1	2.8	48.9	3.1	65.0
5.0	44.8	14.8	67.0	14.2	66.2	7.1	68.4	12.3	73.0
10.0	89.6	22.5	74.9	27.4	69.5	10.0	76.1	19.3	78.0
15.0	134.4	27.9	79.3	37.8	71.8	12.3	80.2	31.8	76.0
20.0	179.2	31.9	82.2	48.2	73.1	14.5	83.0	40.5	77.0

TABLE 2. Storage (in Mbytes) and compression percentage (%C) for a space of size $32,768 \times 32,768$.

		Uniform				Gaussian			
Density	iKd-tree	k^2-tree		cKd-tree		k^2-tree		cKd-tree	
%	MB	MB	%C	MB	%C	MB	%C	MB	%C
0.1	3.8	2.9	23.5	1.9	50.1	2.1	44.5	1.7	56.3
0.5	19.2	11.0	42.7	8.2	57.5	7.1	62.9	7.0	63.7
1.0	38.4	18.9	50.9	15.0	60.9	11.4	70.4	12.6	67.2
5.0	192.0	59.1	69.2	60.6	68.4	28.6	85.1	49.2	74.4
10.0	384.0	89.9	76.6	109.4	71.5	39.9	89.6	90.3	76.5



a) Space of size $16,384 \times 16,384$



b) Space of size $32,768 \times 327,68$

FIGURE 5. Data Compression Percentages.

emerges wherein the values of c_p and c_q remain identical in every consecutive pair of recursive calls. Consequently, this pattern forces the algorithm to explore a substantial portion of the implicit tree. In such instances, the time complexity is upper-bounded by $O(n \cdot \log m)$.

Algorithm *RangeQuery()* (refer to Algorithm 4) descends along the implicit tree until it encounters an internal node i, where the partition intersects with the given range query rQ. This internal node i has a height denoted as $h \leq \log_2 n$, which is contingent on the range query rQ. During each recursive call, the algorithm assesses whether both partitions intersect with the range query rQ and subsequently processes those specific partitions.

Consider only the partitions of the leaf nodes, denoted as $R_l = R_{l1}, R_{l2}, \ldots, R_{lk}$, where k equals $(n+1)/2$ in the case of a full tree. If a partition R_{lj} intersects with the range query rQ, the leaf needs to be queried, regardless of whether the leaf is within rQ or not. Every internal node marked for querying is part of the path of a node whose partition intersects with rQ. Consequently, the time complexity of Algorithm *RangeQuery()* is bounded by the number of intersected partitions and the time needed to query them, yielding an upper bound of $O(|rQ \cap R_l| \cdot h \cdot \log m + (\log n - h))$.

However, it is worth noting that the leaves involved in this intersection share internal nodes. Since each internal node divides its partition into two, and considering that each partition is in close proximity to its sibling partition, an improvement in the estimated time complexity can be derived. In this case, the time behavior is better described as $O(|rQ \cap R_l| \cdot \log m + (\log n - h))$.

V. EXPERIMENTS

In this section, we conduct a series of experiments to analyze the performance of the compact data structure cKd-tree in terms of compression (Section V-A) and query

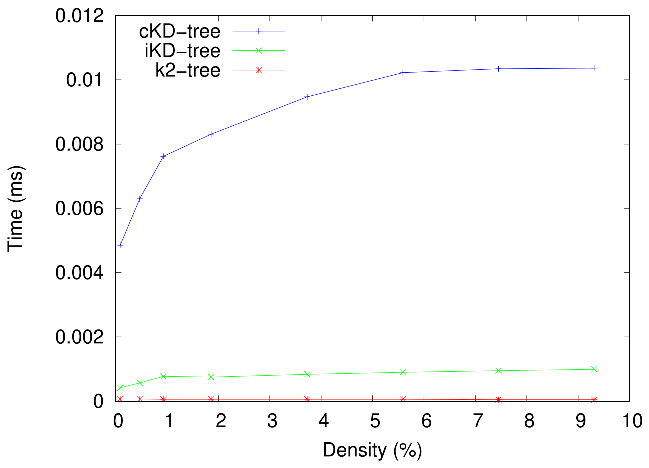


FIGURE 6. **Point query performance on a 32, 768 × 32, 768 matrix with Uniform distribution.**

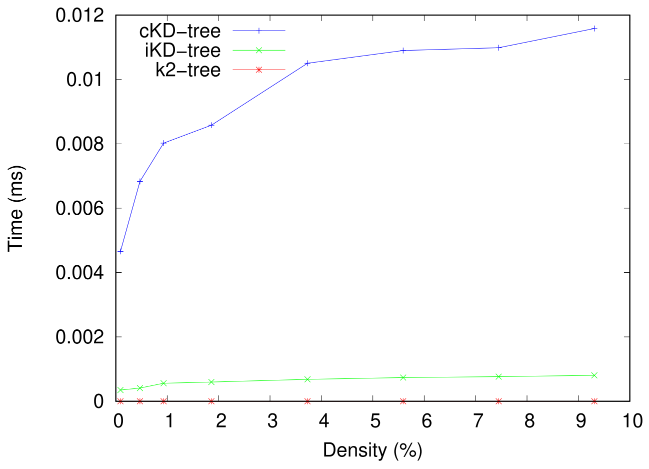


FIGURE 7. **Point query performance on a 32, 768 × 32, 768 matrix with Gaussian distribution.**

execution time (Section V-B). We compare cKd-tree against iKd-tree and k^2-tree, with the latter being known for its efficiency in compression and query execution according to existing literature. The iKd-tree and cKd-tree structures were implemented in C++, while we utilized the k^2-tree implementation from [22].[1]

The experiments were conducted on a server running the Linux operating system (Ubuntu 22.04.3 LTS) equipped with an Intel(R) Xeon(R) E3-1220 V2 CPU, featuring 4 cores operating at 3.1 GHz each, 24 GB of RAM, and a 1 TB SATA 7.2 k hard drive without RAID configuration.

In the experiments, we utilized synthetic two-dimensional datasets comprising points. The dataset sizes ranged from 250, 000 to 100, 000, 000, distributed according to both Uniform (indicated as (U) in the charts) and Gaussian (indicated as (G) in the charts) distributions in space. The datasets were generated within spaces (matrices) of dimensions $2^{14} \times 2^{14}$ and $2^{15} \times 2^{15}$.

[1]Available at https://github.com/simongog/sdsl-lite

A. COMPRESSION AND STORAGE

The compression percentage, or simply compression, for a set of points $S \subseteq \mathbb{N}^2$ located in an array of $m \times m$ and a data structure cKd-tree or k^2-tree, is defined as:

$$Compress = 100 - 100 \cdot (su/ik)$$

Here, su is the storage in bits that are required by the data structure cKd-tree or k^2-tree. The variable $ik = 2 \cdot |S| \cdot \lceil \log_2 m \rceil$ represents the storage (bits) required by iKd-tree considering an optimal number of bits to represent the coordinates of the points. On the other hand, the density of the set of points S is given by:

$$Density = 100 \cdot |S|/m^2$$

These formulas provide a straightforward way to calculate the compression and density metrics for point sets using cKd-tree or k^2-tree in comparison to the reference iKd-tree.

In this section, we examine the following percentage density levels: 0.10%, 0.50%, 1.0%, 5.0%, 10.0%, 15.0% and 20.0%, with array dimensions set to $m = 16,384$ and $m = 32,768$.

Tables 1 and 2 present the total storage of iKd-tree, along with the compression percentages of cKd-tree and k^2-tree, considering sets with Uniform and Gaussian distribution and densities ranging from 0.1% to 20.0% for a space of size 16, 384 × 16, 384 (Table 1) and densities between 0.1% and 10.0% for a space of size 32, 768 × 32, 768 (Table 2).

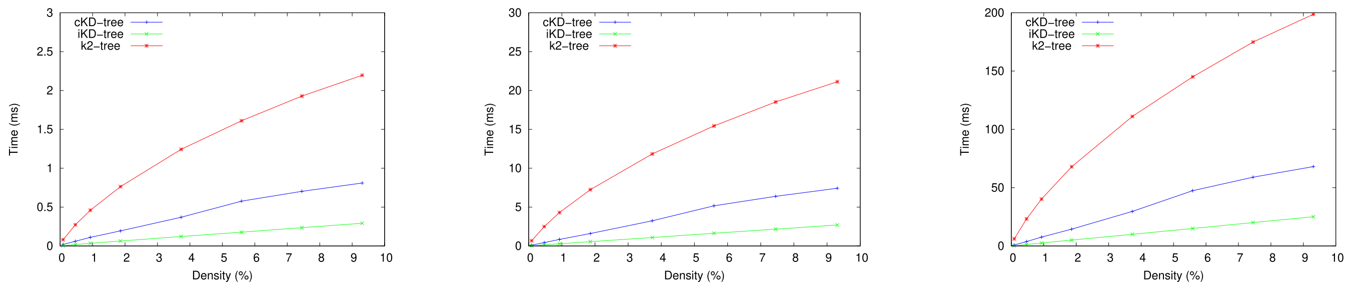
Figure 5 illustrates the compression percentages for both cKd-tree and k^2-tree compact data structures in two space scenarios (sizes 16, 384 × 16, 384 and 32, 768 × 32, 768, respectively). The structures exhibit competitive compression percentages ranging from approximately 18% to 90%. This compression performance improves as the density increases. At higher densities ($\geq$ 12) and for Gaussian distribution (G in the charts), k^2-tree marginally outperforms cKd-tree. Conversely, at lower densities ($\leq$ 6), cKd-tree outpaces k^2-tree.

The enhanced compression performance of cKd-tree at higher densities can be attributed to the spatial proximity of points. In such scenarios, the resulting spiral codes of the points are generally smaller, leading to a highly compressible sequence of integers by DACs.

B. EXECUTION TIME OF QUERIES

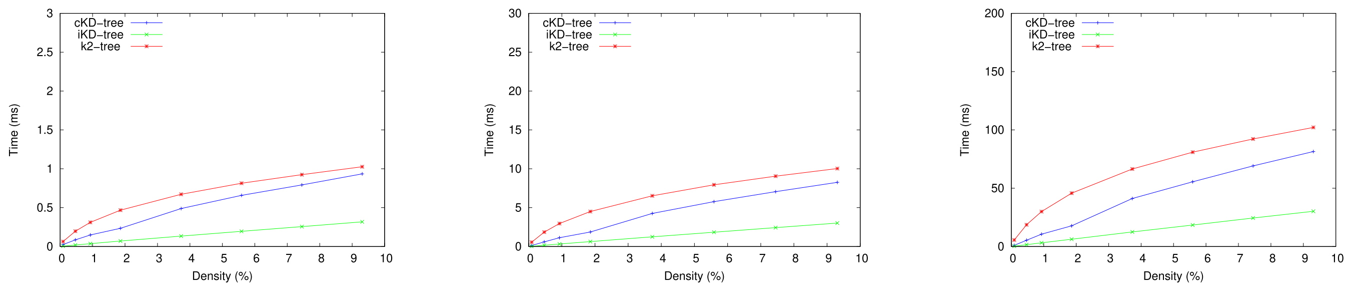
We assess both point query and range query operations across the three data structures. Regarding point query, we measure the average execution time for 1, 000 queries, divided evenly between successful queries (where the queried point is present in the set) and unsuccessful queries (where the point is not in the set). For range query, we consider ranges representing 0.01% (328 × 328), 0.1% (1, 036 × 1, 036), and 1.0% (3, 276 × 3, 276) of the total matrix size (global space). The execution time for each range query is averaged over 1, 000 rectangles uniformly distributed within the matrix.

Concerning the range query, the outcome is not a list of points but the count of points that intersect with the specified



Range of 328×328 Range of $1,036 \times 1,036$ Range of $3,276 \times 3,276$

FIGURE 8. **Range query performance on a 32, 768 × 32, 768 matrix with Uniform distribution.**



Range of 328×328 Range of $1,036 \times 1,036$ Range of $3,276 \times 3,276$

FIGURE 9. **Range query performance on a 32, 768 × 32, 768 matrix with Gaussian distribution.**

range. Nevertheless, each data structure individually traverses the points constituting the answer, and the extra cost associated with storing the points in the answer remains consistent across all structures.

C. POINT QUERY

Figure 6 displays the execution times of three structures in solving the point query for a Uniform distribution of points. As expected, k^2-tree outperforms cKd-tree by a significant margin (about 200 times faster), and iKd-tree also performs faster (11 times) than cKd-tree. The advantage of k^2-tree is attributed to its $O(\log_2 m)$ query execution time, while iKd-tree avoids the cost of decompressing points stored with DACs.

In Figure 7, the execution times for point query are depicted, considering a Gaussian distribution of points. Notably, the differences observed in point query execution times are consistent with those observed in the Uniform distribution case (Figure 6).

D. RANGE QUERY

Figure 8 illustrates the execution times of range query over a matrix of size $32,768 \times 32,768$ for different range sizes. Notably, cKd-tree significantly outperforms k^2-tree, particularly for the range of $3,276 \times 3,276$ where it achieves a speedup of approximately 4.7 times. And, as expected, iKd-tree outperforms cKd-tree in range query, albeit to a lesser extent, with an average speedup of about 3 times for range queries of size $3,276 \times 3,276$.

Figure 9 illustrates the execution times for the range query considering a Gaussian distribution. Interestingly, in this distribution, k^2-tree has better results, but it is still outperformed by cKd-tree, in a lesser extend. The disparities between cKd-tree and iKd-tree remain consistent with those observed in the Uniform distribution. The improved performance of k^2-tree in this distribution is attributed to its optimal operation when points are concentrated in specific areas of space. This characteristic makes k^2-tree sensitive to point distribution, whereas both iKd-tree and cKd-tree exhibit similar behavior across different distributions.

VI. CONCLUSION

This article introduces a novel compact data structure named cKd-tree designed for representing an implicit static Kd-tree. Through experimental evaluations against k^2-tree and iKd-tree, our findings reveal that cKd-tree achieves compression rates ranging from 45% to 77% depending on the density, exhibiting superior compression at higher densities. In comparison with k^2-tree, cKd-tree outperforms when densities are below 5%, compressing up to twice as much. However, at higher densities (above 5%), both structures demonstrate competitive performance.

In terms of execution time for point queries, cKd-tree exhibits a slower performance compared to k^2-tree, as expected. However, cKd-tree showcases superior execution times for range queries, significantly outperforming k^2-tree, especially in scenarios with Uniform distribution where cKd-tree is approximately 4.7 times faster. This performance gap continues to widen with increasing data density.

Furthermore, while our proposal is currently implemented in two dimensions, its natural extension to higher dimensions is feasible. The spiral encoding in d dimensions involves a concentric numbering of size $dist^d$, where $dist$ represents the distance between points. Consequently, the extension to a d dimensional iKd-tree involves encoding the children of each node using d dimensional spiral encoding.

The Kd-tree is a widely adopted multidimensional/spatial data structure applied across various applications, with numerous geometric algorithms tailored for tasks like nearest neighbor searches and range queries. cKd-tree extends this versatility by providing a competitive alternative, enabling the direct implementation of these algorithms in a compact form. As evidenced by our experiments, cKd-tree exhibits superior performance compared to k^2-tree, particularly when querying aggregate data. This suggests its potential for enabling faster execution on more intricate queries, including but not limited to K nearest neighbors and Pareto set calculations.

In comparison to the Kd-tree, cKd-tree stands out by achieving similar functionality while utilizing only 30% of the storage. This efficiency is particularly valuable for sets of certain sizes, where cKd-tree helps circumvent I/O operations that are orders of magnitude slower than direct memory access.

Our proposal, as per the reviewed literature, marks the first attempt to introduce a compact version of a Kd-tree. The reduced storage demands of cKd-tree position it as a viable choice for applications on devices with constrained storage capacities, such as tablets and mobile phones.

REFERENCES

[1] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, no. 9, pp. 509–517, Sep. 1975.

[2] H. Fuchs, Z. M. Kedem, and B. F. Naylor, "On visible surface generation by *a priori* tree structures," *ACM SIGGRAPH Comput. Graph.*, vol. 14, no. 3, pp. 124–133, Jul. 1980.

[3] H. Samet, "The quadtree and related hierarchical data structures," *ACM Comput. Surv.*, vol. 16, no. 2, pp. 187–260, Jun. 1984.

[4] V. Gaede and O. Günther, "Multidimensional access methods," *ACM Comput. Surv.*, vol. 30, no. 2, pp. 170–231, Jun. 1998.

[5] H. Samet, *Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling)*. San Francisco, CA, USA: Morgan Kaufmann Publishers, 2005.

[6] R. A. Brown, "Building a balanced k-d tree in $o(kn \log n)$ time," *J. Comput. Graph. Techn.*, vol. 4, no. 1, pp. 50–68, Mar. 2015.

[7] 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.

[8] R. Grossi, A. Gupta, and J. S. Vitter, "High-order entropy-compressed text indexes," in *Proc. 40th Annu. ACM-SIAM Symp. Discrete Algorithms*, Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2003, pp. 841–850.

[9] G. Navarro, "Wavelet trees for all," *J. Discrete Algorithms*, vol. 25, pp. 2–20, Mar. 2014.

[10] R. Grossi and J. S. Vitter, "Compressed suffix arrays and suffix trees with applications to text indexing and string matching," *SIAM J. Comput.*, vol. 35, no. 2, pp. 378–407, Jan. 2005.

[11] K. Sadakane, "Compressed text databases with efficient query algorithms based on the compressed suffix array," in *Proc. 11th Int. Conf. Algorithms Comput.* Berlin, Germany: Springer, 2000, pp. 410–421.

[12] P. Ferragina and G. Manzini, "Opportunistic data structures with applications," in *Proc. 41st Annu. Symp. Found. Comput. Sci.*, Nov. 2000, pp. 390–398.

[13] N. R. Brisaboa, M. R. Luaces, G. Navarro, and D. Seco, "Space-efficient representations of rectangle datasets supporting orthogonal range querying," *Inf. Syst.*, vol. 38, no. 5, pp. 635–655, Jul. 2013.

[14] J. F. Castro, M. Romero, G. Gutiérrez, M. Caniupán, and C. Quijada-Fuentes, "Efficient computation of the convex hull on sets of points stored in a k-tree compact data structure," *Knowl. Inf. Syst.*, vol. 62, no. 10, pp. 4091–4111, Oct. 2020.

[15] F. Santolaya, M. Caniupán, L. Gajardo, M. Romero, and R. Torres-Avilés, "Efficient computation of spatial queries over points stored in k2-tree compact data structures," *Theor. Comput. Sci.*, vol. 892, pp. 108–131, Nov. 2021.

[16] G. Navarro, *Compact Data Structures—A Practical Approach*. Cambridge, U.K.: Cambridge Univ. Press, 2016.

[17] N. R. Brisaboa, A. Gómez-Brandón, G. Navarro, and J. R. Paramá, "GraCT: A grammar-based compressed index for trajectory data," *Inf. Sci.*, vol. 483, pp. 106–135, May 2019.

[18] N. R. Brisaboa, S. Ladra, and G. Navarro, "DACs: Bringing direct access to variable-length codes," *Inf. Process. Manage.*, vol. 49, no. 1, pp. 392–404, Jan. 2013.

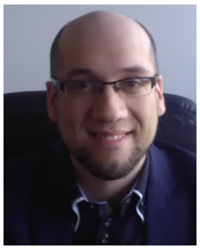
[19] C. E. Sanjuan-Contreras, G. G. Retamal, M. A. Martínez-Prieto, and D. Seco, "CBiK: A space-efficient data structure for spatial keyword queries," *IEEE Access*, vol. 8, pp. 98827–98846, 2020.

[20] S. Ladra, J. R. Paramá, and F. Silva-Coira, "Scalable and queryable compressed storage structure for raster data," *Inf. Syst.*, vol. 72, pp. 179–204, Dec. 2017.

[21] F. Claude and G. Navarro, "Practical rank/select queries over arbitrary sequences," in *Proc. 15th Int. Symp. String Process. Inf. Retr.*, 2009, pp. 176–187.

[22] S. Gog, T. Beller, A. Moffat, and M. Petri, "From theory to practice: Plug and play with succinct data structures," in *Proc. 13th Int. Symp. Experim. Algorithms (SEA)*, 2014, pp. 326–337.

GILBERTO GUTIÉRREZ received the Ph.D. degree in computer science from Universidad de Chile, in 2007. From 2010 to 2012, he was a member of the board of directors of the Chilean Computer Science Society. He is currently an Associate Professor with the Departamento de Ciencias de la Computación y Tecnologas de Información, Universidad del Bío-Bío. His research interests include spatial and temporal databases, data structures, and algorithms.



RODRIGO TORRES-AVILÉS received the Ph.D. degree in applied mathematics from 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.

MÓNICA CANIUPÁN received the Ph.D. degree in computer science from Carleton University, Ottawa, ON, Canada, in 2007. She is currently a Full Professor with the Departamento de Sistemas de Información, Universidad del Bío-Bío. Since 2020, she has been a member of the board of directors of the Chilean Computer Science Society. Her research interests include databases, data consistency, data warehousing, compact data structures, and spatio-temporal databases.