Received May 11, 2020, accepted May 18, 2020, date of publication May 25, 2020, date of current version June 5, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2997258

*c*B*i*K: A Space-Efficient Data Structure for Spatial Keyword Queries

CARLOS E. SANJUAN-CONTRERAS[1], GILBERTO GUTIÉRREZ RETAMAL[1], MIGUEL A. MARTÍNEZ-PRIETO[2], AND DIEGO SECO[3]

[1]Department of Computer Science and Information Technology, University of Bío-Bío, Chillán 3780000, Chile
[2]Department of Computer Science, University of Valladolid, 40005 Segovia, Spain
[3]Department of Computer Science, Millenium Institute for Foundational Research on Data (IMFD), Universidad de Concepción, Concepción 4070386, Chile

Corresponding author: Carlos E. Sanjuan-Contreras (csanjuan@ubiobio.cl)

The work of Carlos E. Sanjuan-Contreras and Gilberto Gutiérrez R. was supported by the University of Bío-Bío under Grant 192119 2/R and Grant 195119 GI/VC. The work of Miguel A. Martínez-Prieto was supported by the Ministerio de Economía y Competitividad (Presupuestos Generales del Estado (PGE) and Fondo Europeo de Desarrollo Regional (FEDER)), Spain, under Grant Datos 4.0: TIN2016-78011-C4-1-R. The work of Diego Seco was supported in part by the Millennium Institute for Foundational Research on Data, and in part by the National Agency for Research and Development (ANID) through Fondo Nacional de Desarrollo Científico y Tecnológico (FONDECYT), Chile, under Grant 1170497.

ABSTRACT A vast amount of geo-referenced data is being generated by mobile devices and other sensors increasing the importance of spatio-textual analyses on such data. Due to the large volume of data, the use of indexes to speed up the queries that facilitate such analyses is imperative. Many disk resident indexes have been proposed for different types of spatial keyword queries, but their efficiency is harmed by their high I/O costs. In this work, we propose *c*B*i*K, the first spatio-textual index that uses compact data structures to reduce the size of the structure, hence facilitating its usage in main memory. Our experimental evaluation, shows that this approach needs half the space and is more than one order of magnitude faster than a disk resident state-of-the-art index. Also, we show that our approach is competitive even in a scenario where the disk resident data structure is *warmed-up* to fit in main memory.

INDEX TERMS Bitmap, compact data structure, kNN, spatial keyword query, spatio-textual data.

I. INTRODUCTION

The boom of mobile devices has made it possible to live experiences and to access services that were inconceivable a few years ago. Location-based applications are a good example. Although they are ubiquitous nowadays, a decade ago it was hard to imagine that a mobile phone would allow us to look for the nearest ''tapas'' restaurant or to obtain a list of pet-friendly hotels located less than 3 km away. These applications have emerged due to the development of positioning technologies such as GPS *(Global Positioning System)*, but also due to the consolidation of geo-tagged datasets, which provide textual descriptions (usually as lists of keywords) for geo-positioned objects.

SPOI[1] *(Smart POI dataset)* is a good example of geo-tagged dataset. It contains more than 27 million points of interest around the world, including cafes, pubs, restaurants, or hotels, which are described using keywords about their specialties or the amenities they provide. Picture datasets or collections of microposts from social-networks can be also considered geo-tagged datasets. The former provides large collections of geo-positioned pictures that are described using particular keywords (e.g. an image of the *''Grand Canyon''* has its GPS coordinates, and also a set of keywords like *''Natural park''*, *''USA''*, or *''Ancestral Puebloans''*, among others). On the other hand, collections from social networks expose large amounts of microposts that contain the location from which they were published, and lists of hashtags, as keywords.

In other domains, such as the Web, where just a portion of the content is geo-tagged, there are some attempts to automatically detect locations from web resources [1]. Such results would enable the development of general location-aware search engines, in which data structures such as the one proposed in our work, are essential for an efficient information retrieval as an extension of ubiquitous inverted indexes.

Geo-tagged datasets are increasingly larger, hence managing and querying them is becoming more challenging. Many and varied approaches have been described in the state

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

[1]https://sdi4apps.eu/spoi/

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

of the art for such purposes, but all of them share a main feature: they are designed over disk-resident data structures (indexes), which involve an important overhead in query resolution time due to I/O operations. More recently, *compact data structures* [2] have emerged as an efficient approach for managing large datasets in main memory. These structures store data in a compact manner, and are able to query them with no prior decompression. This approach avoids costly I/O operations, at the price of performing more computations to access the data. In other words, compact data structures are usually slower than traditional approaches when running in the same level of the memory hierarchy, but they are more likely to fit in higher levels, drastically increasing query time performance. Although compact data structures have been successfully used for different applications, including spatial [3] or keyword-based search [4], they have not been evaluated yet for queries that involve spatial and keyword-based predicates, to the best of our knowledge.

In this paper, we propose a novel compact data structure (called *c*B*i*K) that is able to store geo-tagged data in compact space and it is also able to resolve three different spatial keyword queries, Boolean Top-*k* Spatial Keyword Query (B*k*SKQ), Ranked Top-*k* Spatial Keyword Query (R*k*SKQ) and Boolean Range Searching Spatial Keyword Query (*b*RS-SKQ). *c*B*i*K indexes spatial objects using an implicit KD-tree, with no pointers, and encodes their descriptive keywords using compressed bitmaps. Our experiments report that *c*B*i*K uses only $35-40\%$ of the space required by a state-of-the-art index, while answers the corresponding queries up to 2 orders of magnitude faster for a selected testbed, which includes different real-world datasets. When both indexes reside in main memory, query times are comparable, which is not a surprising result when using compact data structures as explained above. Furthermore, it is worth noting that *c*B*i*K is a static index; i.e. it must be rebuilt from scratch to add new data or update spatio-textual objects descriptions. This is common when using compact data structures due to the cost of update bitmaps [2].

A preliminary partial version of this research was presented in [5]. In this article, we describe an improved version of our data structure, which is also able to answer R*k*SKQ and *b*RS-SKQ queries. In addition, we provide a more comprehensive experimental evaluation, including some new experiments to thoroughly evaluate the performance of our data structure and algorithms.

The rest of the paper is organized as follows. First, Section II formally states the problem of spatio-textual indexing and its variants. Then, Section III describes common approaches for dealing with spatial and temporal data independently. Also, some basic concepts about compact data structures are provided. This section can be skipped by the reader with previous knowledge about such topics. Section IV summarizes state-of-the-art approaches to combine both dimensions in spatio-textual indexes. Then, Section V describes our data structure, and Section VI explains the three algorithms proposed to implement the

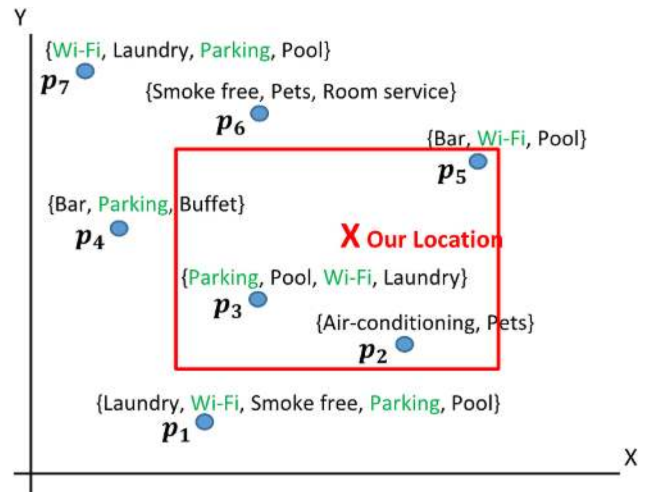

FIGURE 1. Example of a geo-tagged dataset describing hotels and the services each provides.

corresponding spatial keyword queries. A comprehensive experimental evaluation is presented in Section VII, where *c*B*i*K is compared to a reference approach. Finally, Section VIII concludes about our current results and devises future lines of research.

II. PROBLEM DEFINITION

Before delving into the background of the paper, two basic definitions need to be established:

Definition 1 (Spatio-Textual Object): A spatio-textual object o is a tuple $\langle l, t\rangle$, where $o.l$ provides the corresponding spatial position, in a two dimensional plane, and $o.t$ is the list of keywords that describe the object.

Definition 2 (Geo-Tagged Dataset): A geo-tagged dataset D is a collection of n spatio-textual objects, which are tagged using a set T of m different keywords.

Fig. 1 illustrates a simple geo-tagged dataset that contains $n = 7$ spatio-textual objects, each one describing a hotel and the set of different services it offers. Note that $m = 10$ different services are offered, and the corresponding set of keywords is T = *{Air-conditioning, Bar, Buffet, Laundry, Parking, Pets, Pool, Room-service, Smoke-free, Wi-Fi}*.

A. SPATIAL KEYWORD QUERIES

A *Spatial Keyword Query* (SKQ) takes a user location and user-supplied keywords as arguments and returns objects that are spatially and textually relevant to these arguments [6]. Although different SKQs have been proposed in the state of the art, we focus on the three types studied in [7], that are explained below.

1) BOOLEAN TOP-*k* SPATIAL KEYWORD QUERY (B*k*SKQ)

B*k*SKQ retrieves the k objects closest to a given location, which also satisfy the requested keywords. More formally, B*k*SKQ is defined as a query $q = \langle l, t, k\rangle$, where $q.l$ is the location of the query point (latitude and longitude), $q.t$ is the

list of requested keywords, and $1 \leq q.k \leq n$ is the maximum number of objects to be retrieved.

BkSKQ returns a result set that contains, at most, $q.k$ objects: $\{o_1, o_2, \ldots, o_{q.k}\}$, which are the $q.k$ objects closest to $q.l$, ordered by Euclidean distance (ascending order), that also satisfy $q.t \subseteq o_i.t$, with $1 \leq i \leq q.k$.

An example of BkSKQ query is *looking for the 2 closest hotels to our location*[2] *that provide Wi-Fi and Parking services*: $q = \langle (x_i, y_i), \{\text{Wi-Fi, Parking}\}, 2\rangle$. As can be seen in Fig. 1, p_3 and p_1 are the closest hotels to our location that provide the requested services, so they are returned in such an order.

2) RANKED TOP-k SPATIAL KEYWORD QUERY (RkSKQ)

RkSKQ uses a function score to retrieve the best-rated objects according to their proximity to the desired location and their textual relevance to the requested keywords. Specifically, the function to obtain the score of a spatio-textual object o for a query q, is defined as:

$$\tau(o, q) = \alpha \cdot \delta(o.l, q.l) + (1 - \alpha) \cdot \theta(o.t, q.t) \tag{1}$$

Note that $\delta(o.l, q.l)$ corresponds to the spatial proximity between o and q, and $\theta(o.t, q.t)$ measures the textual relevance between o and q. Spatial proximity and textual relevance are weighted by a preference parameter $\alpha \in [0, 1]$. Thus, if $\alpha = 1$, $\tau(o, q)$ only considers spatial proximity, while if $\alpha = 0$, only textual relevance is used to rank the best candidates.

More formally, RkSKQ is defined as a query $q = \langle l, t, k, \alpha\rangle$, where $q.l$, $q.t$, and $q.k$ have the same meaning that in BkSKQ, and $\alpha \in [0, 1]$ weights the importance of spatial proximity and textual relevance in the final result. Thus, RkSKQ returns a ranked list that includes the best-rated spatial points (in descending order) for the given query.

a: SPATIAL PROXIMITY

The spatial proximity $\delta(o.l, q.l)$ is calculated using the *normalized Euclidean distance*, as defined in (2), where $dist(o.l, q.l)$ is the Euclidean distance between o and q, and $dist_{max}$ is the maximum Euclidean distance between two objects in D:

$$\delta(o.l, q.l) = 1 - \frac{dist(o.l, q.l)}{dist_{max}} \tag{2}$$

b: TEXTUAL RELEVANCE

Different information retrieval models, like Language Model [8], Cosine Similarity [9] or BM25 [10], have been used to obtain the textual relevance of a point to a given query. In our case, we use a simple model that assigns the same relevance to all the keywords requested in a query, so the textual relevance of a point (to a given query) is proportional to the number of requested keywords that it contains:

$$\theta(o.t, q.t) = \sum_{i=1}^{|q.t|} \frac{1}{|q.t|}, \quad \text{if } q.t_i \in o.t \tag{3}$$

An example RkSKQ query, where proximity is considered more relevant than the services provided by the hotel (e.g. proximity is weighted as 0.75) is *looking for the 2 best hotels that provide Wi-Fi and Parking services*: $q = \langle (x_l, y_l), \{\text{Wi-Fi, Parking}\}, 2, 0.75\rangle$. In this case, p_3 is returned because it provides both services and it is closer to our location. However, p_5 is returned (instead of p_1) although it does not provide Parking, because it is closer to our location, and proximity is more relevant for the given query.

3) BOOLEAN RANGE SEARCHING SPATIAL KEYWORD QUERY (bRS-SKQ)

bRS-SKQ retrieves all objects located within a determined query region that also contain all the requested keywords. More formally, $q = \langle r, t\rangle$ where $q.r$ indicates a (rectangular shaped) query region and $q.t$, as in the previous queries, provides the list of requested keywords.

The result of bRS-SKQ is a collection of objects: $\{o_1, \ldots\}$, such that $\forall o_i$, it is satisfied that $o_i.l \cap q.r \neq \emptyset \wedge q.t \subseteq o_i.t$.

An example of bRS-SKQ is *looking for hotels that provide Wi-Fi in our region*[3], which is represented by the red-line square in Fig. 1: $q = \langle (x_{bl}, y_{bl}; x_{tr}, y_{tr}), \{\text{Wi-Fi}\}\rangle$. In this case, only *p3* and *p5* are returned, because the other candidates with Wi-Fi are outside the given region.

III. PREVIOUS CONCEPTS

In this section, we summarize the basic knowledge about spatial indexes, textual indexes and compact data structures that is necessary to follow our work. The reader with previous knowledge about such topics may safely skip the section.

A. SPATIAL INDEXES

A spatial index is a data structure designed to answer different types of spatial queries. Here we review three of the most used spatial indexes, which are also components of the spatio-textual indexes described in Section IV.

1) R-TREE

The R-Tree is one of the most studied spatial access methods and also one of the most widely used in practice, as it has been adopted by several Database Management Systems (DBMS) such as Oracle and Postgres. It was proposed by Güttman [11] to dynamically index multidimensional information (points, polylines and regions in space) and it performs a recursive partition of the space where spatial objects are located and organizes such partitions into a balanced tree.

In general, leaf nodes of the R-tree store a simplification of the actual objects called *Minimum Bounding Rectangle* (MBR). Leaves also store a reference *ref* to the actual object.

[2] We assume that *"our location"* is defined by coordinates (x_i, y_i).

[3] We assume that the region is defined by its bottom-left and top-right coordinates: (x_{bl}, y_{bl}) and (x_{tr}, y_{tr}), respectively.

In the particular case of points, the leaves of the tree can store the actual objects. Internal nodes, on the other hand, correspond to a disk block and contain entries of the form $\langle MBR_i, ref_i \rangle$, where ref_i is a pointer to the corresponding child node, and MBR_i is the smallest rectangle that covers all the MBRs associated with the child nodes. The key idea behind the R-Tree is to store spatially-close objects in the same block.

Although the R-Tree was initially proposed to solve range queries, different algorithms have been proposed to solve varied geometric problems. For example, the $k > 0$ nearest neighbors to a given point q [12], the $k > 1$ pairs of nearest neighbors between two sets of points [13], or the computation of the Hausdorff distance between two sets of points [14], to name a few.

2) QUADTREE

A Quadtree [15], [16] is a multidimensional data structure used to represent and index point-type objects in a d-dimensional space. Like the R-Tree, it recursively decomposes the space by means of iso-oriented hyperplanes representing the subspaces using a tree data structure, whose root represents the entire space.

In a space of d dimensions, each internal node of the quadtree has 2^d children, each one representing the 2^d subspaces of the space associated with the node. The recursive decomposition continues until the number of objects is below a threshold. A quadtree is not necessarily balanced because the denser subspaces (i.e. those containing more objects) can generate nodes much deeper than less dense subspaces [16]. This is because, unlike the R-Tree, the space partitioning is oblivious of the actual objects and just depend on the universe (i.e. the space in which the objects are represented).

3) KD-TREE

The KD-Tree [17] was proposed to index points in k-dimensions. It also decomposes the space recursively but, unlike the Quadtree, the division does not have to generate two partitions of equal size. An orthogonal hyperplane aligned with the axes of coordinates is used in the partitioning. In each level, the axes alternate; that is, if working with two dimensions, the plane is first cut by the x axis, then by the y axis, and in the next level again by the x axis.

The structure is represented as a binary tree in which each internal node represents a cut in space. To balance the tree, the points that are lower than the current cut-off point, which is the median of the subspace in the order of the corresponding dimension, are stored in the left sub-tree and the others in the right sub-tree. The leaf nodes represent the point themselves when there are no more subspaces to divide.

B. TEXT INDEXES

A text index is a data structure particularly designed to answer text-based queries. Here, we introduce text indexes that are frequently used to perform efficient keyword-based queries.

1) INVERTED INDEX

An inverted index (also referred to as *inverted file*) [18] is the reference index for information retrieval purposes. It is built over the set of m different words used in a data collection. An inverted index is composed by two main components (i) a *vocabulary*, which encodes the set of m words, and (ii) a set of m *inverted lists* (or *posting lists*), where each one stores references to all occurrences of a particular word.

Posting lists may encode occurrences information at different levels of granularity. According to the scope of this paper, we consider that each list stores the IDs of the spatio-textual objects that are described by a given keyword.

Inverted indexes excel at word-based query resolution. They basically look for the searched word in the vocabulary, and the corresponding inverted list is retrieved. Boolean AND/OR queries, like the ones required by SKQs, are also resolved efficiently by manipulating the inverted lists of each requested keyword.

2) SIGNATURE FILE

A signature file is a text indexing approach [19] that divides a text collection into logical blocks of D distinct words. Each word has its signature, a bitstring of size F, whith m bits activated. F and m are parameters of the file. To encode a block, the signatures of their words are superimposed (bit OR-ed) to obtain a single F-bit pattern. The whole text collection is encoded by concatenating block signatures.

The process is replicated for query resolution. That is, the signatures of the requested words are first superimposed and the resulting signature is searched.

In comparison with inverted indexes, signature files require much less storage space but, on the other hand, have a high processing cost to perform a query due to the I/O operations [20].

C. COMPACT DATA STRUCTURES

Compact data structures store data using small space while allow them to be queried with no prior decompression [2]. These structures drastically reduce their memory footprint, enabling to manage large data collections at the top levels of the memory hierarchy, and also avoiding costly I/O operations. Regardless of the type of structure being implemented, most compact data structures are built on top of particular configurations of bitvectors. A *bitvector* is a bit array $B[1, n]$ that provides three main operations:

- `access(B,i)` returns the bit $B[i]$, for any $1 \leq i \leq n$. This operation is similar to that provided by traditional bitmap indexes.
- $\mathtt{rank}_v$`(B,i)` counts the number of times that the bit $v \in [0, 1]$ occurs in $B[1, i]$, for any $0 \leq i \leq n$ (note that $\mathtt{rank}_v$`(B,0)=0`).
- $\mathtt{select}_v$`(B,j)` returns the position of the j-th occurrence of the bit $v \in [0, 1]$ in B, for any $j \geq 0$ (note that $\mathtt{select}_v$`(B,0)=0`).

There are several implementations that provide a space-time trade-off to support such operations. To ensure constant time performance for these operations, it is necessary to enhance bitvectors with some additional structures that add extra bits on top of the bit array. In this paper, we will use the `bit_vector` implementation from SDSL [21], which uses $64\lceil n/64 + 1 \rceil$ bits.

Some other approaches leverage bit redundancy to compress bitvectors, ensuring efficient performance. In this paper, we use an implementation[4] of the *sparse array* (SD-array) [22] to exploit the sparseness of the cBiK bitvectors. This structure excels when the proportion of 1s in the bitvector is below 5%, providing (very) efficient `select(B,1)` in constant time, and `rank(B,1)` in $O(\log(n/m))$, where m is the number of 1-bits.

IV. RELATED WORK

As shown above, different indexes have been proposed to solve spatial and textual queries independently, but solving spatio-textual queries implies the combination of both types of indexes. In this section, we review the main approaches in the literature to solve the three types of queries described in Section II-A.

Most of the proposed indexes focus on efficiently processing the corresponding queries, but do not consider the large storage requirements to achieve such goal. It is worth noting that all approaches described in the following are based on data structures that reside in secondary memory and have large space requirements.

The first approaches to solve bRS-SKQ kept the spatial index apart from the textual index. For example, [23] proposes two approaches using a *Quadtree* and an *Inverted File*. As the indexes are independent, one is used to filter the candidate results, and the other one to refine the result. The main advantage of this approach is its simplicity, but the efficiency is lower when compared with hybrid approaches.

One of the first hybrid approaches was proposed in [24], which combines an R*-tree [25] and an *Inverted File* in different ways. The best results were obtained by the IF-R*, which builds an Inverted File for each term, which is also associated with an R*-Tree containing all the corresponding points. To perform a query, the Inverted File is used to filter the keywords and then, spatial range queries are performed in each candidate R*-tree, which implies additional disk costs due to the access to independent trees. Later, the KR*-tree (Keyword R*-tree) [26] reduced the I/O costs thank to an efficient pruning of the tree traversal.

De Felipe *et al.* [27] described the first approach for BkSKQ queries: the IR^2-Tree, which stores a *signature file* [19] in each node of the tree to summarize the presence/absence of keywords. Our proposal of using compact bitmaps to represent the keywords that are present in each subspace is inspired in such work. However, there are several important differences, such as the use of compact bitmaps (instead of signature files), the use of an implicit KD-tree, and the support for the three types of queries described in Section II-A.

Later, [8] proposed a new kind of top-k query that takes into account both location proximity and text relevancy (RkSKQ), the IR-tree (combines an Inverted File and an R-Tree in which each node is augmented with a summary of the textual information). Similar proposals can be found in [28]–[30].

Rocha-Junior Nørvåg [31] proposed the *Spatial Inverted Index* (S2I) to solve RkSKQ queries using an R-tree and an Inverted File. S2I uses different strategies to index frequent and infrequent keywords. Hence, S2I maps each keyword t to an aggregated R-tree (aR-tree) or to a block storing spatio-textual objects that contain t. The most frequent keywords are stored in the aR-tree using a tree for each keyword. The less frequent ones are stored in blocks in a file, one block per keyword (similar to the Inverted File). In the aR-Tree, each node stores an aggregate value that indicates the maximum impact (in terms of the punctuation of the textual relevance) of the keyword in the objects of the tree. The aR-Tree can be devised as an IR-tree [8], [29] for a single keyword. For queries with a single keyword, only a small tree or a block is accessed, in general. For queries with several keywords, some nodes of a small set of trees or blocks are accessed leading to efficient execution of queries. The potential of S2I is demonstrated in the experimental evaluation of [7], which compares the best 12 spatio-textual indexes to date for the three types of queries described in Section II-A. Although S2I originally responds only RkSKQ, it is easily modifiable to answer the other types of queries. The experimental results show that, in general, it is always among the best indexes for the three types of queries. Taking the S2I as a basic component, some other indexes have been proposed in [32]–[34].

If we classify spatio-textual indexes regarding the text index they use, two main categories arise. On the one hand, [8]–[10], [23], [24], [26], [28], [35], [36] use an Inverted File [18] that orders the objects by their IDs. On the other hand, bitmaps [19] are also used to index textual information contained in the sub-trees. In this case, each bit represents the presence or absence of a keyword in the object. Note that, unlike the ones used in our approach, these are plain bitmaps without support for rank/select operations. The use of sparse bitmaps with support for rank/select operations is a distinguishing idea of our work.

All the approaches described above are based on disk resident data structures and therefore are always penalized by I/O costs. A recent work in [37] evaluates solutions for main memory. However, it focuses on the streaming model. Hence, it adapts the data structures to support data arriving at high speed rates. Also, the temporal dimension plays a more important role in such a model, defining different types of queries from the ones studied in our work.

[4] The `sd_array` implementation from the SDSL library [21].

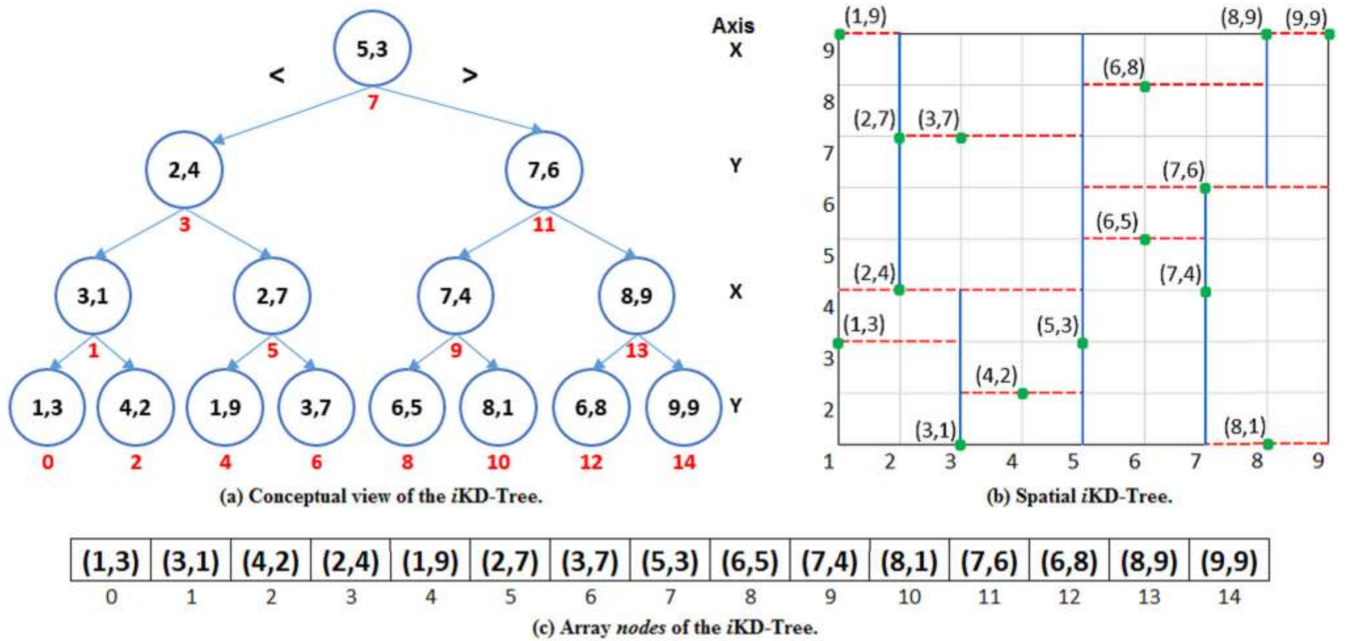

FIGURE 2. Example of a balanced *i*KD-Tree with 15 points. The number of leaves is a power of two for visualization purposes, but this is not a restriction of the data structure.

V. cB*i*K - COMPACT DATA STRUCTURE FOR SPATIO-TEXTUAL DATA

In this section we introduce the data structure cBiK (compact Bitmaps and implicit KD-Tree) to represent spatio-textual datasets. The main components of this data structure are: i) a balanced implicit KD-Tree (iKD-Tree) and ii) two compact bitmaps that represent the keywords. We describe both components below and the query algorithms in Section VI.

A. *i*KD-TREE

The iKD-Tree is based on the KD-Tree proposed in [38] for a static set of points of size n. The KD-Tree is a balanced binary tree that uses pointers to link its nodes. Unlike the KD-Tree, the iKD-Tree does not use pointers and stores the points directly in an array, which we call *nodes*. Like the KD-Tree, the iKD-Tree is a balanced tree and its height is bounded by $O(\log n)$.

To construct the iKD-Tree, the spatial objects are first sorted in each of the dimensions (two in this case). Then, according to the coordinate selected to perform the partition, the element in the middle of the array is the root of the tree. For example, in Fig. 2(a) if x is considered as the partitioning coordinate, the point $(5, 3)$ is stored in the middle of the array (see Fig. 2(c)). Then we proceed recursively with each subarray by alternating the coordinate that guides the subspace partition. Fig. 2(b) shows the partitions of the space generated by the iKD-Tree in Fig. 2(a). The construction of the iKD-Tree takes time $O(dn \log n)$, with d the number of dimensions [38].

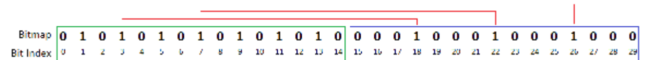

FIGURE 3. Bitmap for the *i*KD-Tree in Fig. 2. The first half, bounded by a green rectangle, identifies the leaf nodes (0) and the internal nodes (1) of the tree. The second half, in blue, indicates all the internal nodes that have an explicit summary.

Associated with the iKD-Tree, the bitmap BM of size $2n$ is used. The first n bits in Fig. 3 indicate whether a node of the iKD-Tree is internal (1) or a leaf (0). The second half of BM indicates if the internal node has an explicit summary (1 if it does, and 0 otherwise) of the T keywords in each of the subspaces generated by the node (see Section V-B2 for more details).

The functional support of the bitmap BM is critical since it helps to build part of the cBiK structure and also facilitates many calculations used in the query algorithms. For example, by using BM it is possible to know, using operation $rank_1(BM, i)$, which internal nodes of the iKD-Tree exist up to a certain position i.

It is possible to traverse the iKD-Tree through the variables *start* and *end*, which store the position in the *nodes* array of the first and last point, respectively, in a subspace generated by the iKD-Tree. If $start = 0$ and $end = n - 1$, then the algorithm is considering the whole space. The position j of the root node of the subspace delimited by *start* and *end* can be obtained as $j = \left\lfloor \frac{start+end}{2} \right\rfloor$. If $access(BM, j) = 0$, then j is a leaf node, otherwise it is an internal node. Left and right children of node j are at positions $\left\lfloor \frac{start+j-1}{2} \right\rfloor$ and $\left\lfloor \frac{j+end+1}{2} \right\rfloor$, respectively.

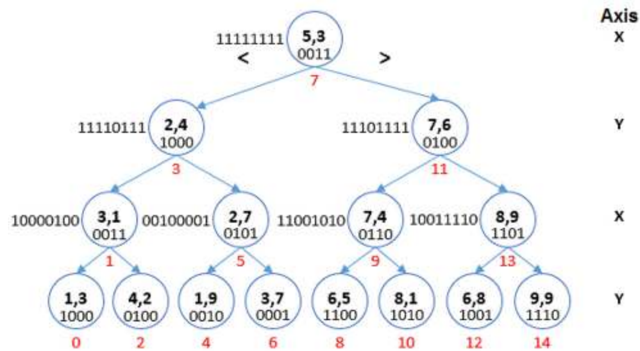

FIGURE 4. Conceptual view of the *i*KD-tree and the bitmaps *BK* (inside the nodes) and *RL* (next to the nodes).

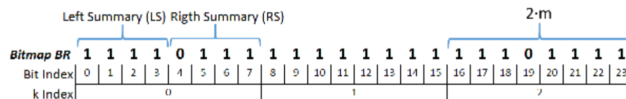

FIGURE 5. Summary Bitmap of the internal nodes with *i* equals 3, 7 and 11 of the *i*KD-Tree in Fig. 4.

B. KEYWORDS REPRESENTATION

Our structure uses two bitmaps, one to represent the keywords of each point and the other to indicate the keywords belonging to the left and right subspaces of the internal nodes of the *i*KD-tree.

1) BITMAP KEYWORDS (*BK*)

BK is a bitmap of size $n \cdot m$ used to indicate the keywords that correspond to each of the nodes or points of the *i*KD-Tree (see the bitmaps inside the nodes in Fig. 4). If the *i*-th keyword is present at a point *j*, with $0 \leq i < m$, the *i*-th bit of *j* is set to 1, and to 0 otherwise. For $0 \leq j < n$, the keywords of node *j* are represented by the bits $[j \cdot m \ldots (j+1) \cdot m - 1]$ of *BK*.

Bitmaps *BK* are encoded using the SD-array [22] to exploit their sparseness. Thus, each bitmap is encoded in space proportional to the number of keywords that describe a particular point, and not to the number of different keywords used in the collection. This ensures the effectiveness of *c*B*i*K even for collections that use a large set of keywords.

2) BITMAP SUMMARY (*BR*)

This bitmap represents the keywords that are in any of the points of each subspace generated by the *i*KD-Tree (see the bitmaps next to each internal node in Fig. 4). Conceptually, each internal node *p* is augmented with a bitmap of size $2 \cdot m$ called local summary (*RL*). The first *m* bits of a bitmap *RL* represent the keywords that are located in any of the points belonging to the subspace to the left of *p* (in our algorithms we refer to these conceptual bitamps as *LS*), and the *m* remaining bits represent the keywords of the points belonging to the subspace to the right (we refer to them as *RS*).

We distinguish two types of internal nodes: i) nodes whose children are leaves, and ii) nodes whose children are internal nodes. Let *p* be an internal node with left child *l* and right child *r*, if *p* is a node of type i), then its *RL* is calculated by concatenating the bits $[l \cdot m \ldots (l+1) \cdot m - 1]$ and $[r \cdot m \ldots (r+1) \cdot m - 1]$ of *BK*. On the other hand, if *p* is of type ii) its construction can be described as an inorder traversal of the sub-tree induced by *p* that recursively accumulates the positions of the keywords in such sub-tree as follows: The first *m* bits of *RL* of *p* are obtained by performing the bitwise *or* operation ($\vee$) between the first *m* bits of the bitmap *RL* associated to *l* with the *m* bits of the bitmap *RL* associated to *r*. The remaining *m* bits are obtained in a similar way. In *BR*, only the *RL*'s of type ii) nodes are stored (in a compact form), as shown in Fig. 5. The *RL*'s of type i) nodes are computed online when necessary.

The procedure to retrieve the bitmap *RL* of an internal node *p* indexed by *j* in the array *nodes* is as follows. First, we need to know the type of node *p*. Let $i_1 = access(BM, j)$ and $i_2 = access(BM, n+j)$. If $i_1 = i_2 = 1$, then *p* is of type ii) and its *RL* is computed as follows. First, we obtain the number of internal nodes of the *i*KD-Tree as $ni = rank_1(BM, n)$, and then we count the number of type ii) nodes as $k = (rank_1(BM, n+j) - ni)$. Finally, the *RL* of node *p* is in the range of positions $[2 \cdot k \cdot m \ldots 2 \cdot m \cdot (k+1) - 1]$, with $k \geq 0$ of bitmap *BR*. If $i_1 = 1$ and $i_2 = 0$, then node *p* is of type i) and its associated *RL* is obtained as explained above.

Note that the *BR* bitmap plays an important role in the processing of spatio-textual queries since it allows the index *c*B*i*K to prune entire branches using both spatial and textual dimensions at the same time. As *BK*, *BR* is also implemented using sparse arrays [22] to optimize the number of bits required to encode each effectively used keyword.

VI. ALGORITHMS FOR SPATIAL KEYWORD QUERIES

Let $q = \langle l, t \rangle$ be a *SKQ* query. Hereinafter, all the algorithms assume a folklore hash-based mapping approach to obtain a numerical identifier for each keyword *t* to be used to access bitmaps *BK* and *BR*. Also, all the algorithms assume that the array *nodes* and the bitmaps *BM*, *BK* and *BR* are global variables.

The traditional approach to find the *k* nearest neighbors in a KD-Tree is used in our solution. That is, from the root of the tree, the algorithm descends to the leaves alternating the axis used for the partition of the subspaces. Both in the descent (leaf nodes) and in the return of the recursion (internal nodes) the points associated with the nodes are used to improve the solution and then, it suffices to determine if the candidate point contains the searched keywords. Parameters like *q* are also considered, which represents the query variable and has a spatial attribute *q.l*, a textual attribute *q.t*, and an attribute *q.k* that indicates the number of requested results; *start* and *end* define the range in *nodes* where to continue the traversal in the *i*KD-Tree; initially $start = 0$ and $end = n - 1$. The variable *depth* indicates the depth in the *i*KD-Tree and determines the direction of the subspace partitions.

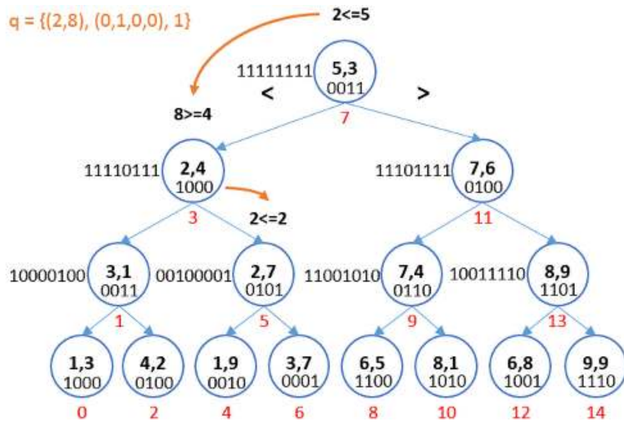

FIGURE 6. B*k*SKQ query example given *q* and the search path on the conceptual tree of the structure *cBiK*. Each node contains the spatial information, the keywords of the point, and the summary bitmap for an efficient pruning.

A. EVALUATION OF B*k*SKQ QUERIES

The general idea of the algorithm is to traverse the tree from the root to the leaves creating and adjusting the solution by simultaneously taking into account the euclidean distance and the searched keywords. To do that, the summary bitmaps are used to determine the subspaces that the algorithm should keep exploring in the traversal. In this way, some complete subspaces can be pruned when the algorithm detects that a searched keyword is not present in the subspace even when there are some points inside it that are close to the query location. Recall that all the results to this type of query must contain all the keywords.

To illustrate the procedure of BkSKQ, we use Fig. 6 with a query q that has three parameters: the query point $(2, 8)$, the keywords that are represented by the bitmap 0100, and the value of $k = 1$.

The query algorithm starts from the root and evaluates the partition in the direction of the x axis. Since the value of the x coordinate of the query point is less than the one of the evaluated point $(2 \leq 5)$, then the left path, to node $(2, 4)$, is followed since its *RL* (the first half of bits) indicates that the keyword searched is present in that subspace. The traversal continues to the point $(2, 7)$, as the value of the query coordinate is less than or equal to the one of the point $(2 \leq 2)$, the algorithm should follow the left side. However, the summary of the point indicates that the keyword is not present in any subsequent subspace, therefore, the traversal is completed and the point is evaluated as a candidate solution. In the return of recursion, the points visited are still evaluated to improve the solution, if appropriate.

Algorithm 1 receives five parameters in addition to those described at the beginning of the section. The variable *heap*, corresponds to a Max-Heap of size k that is used to maintain the candidate points of the solution. At the end of the algorithm, the solution is also stored in *heap*. Note that it may be impossible to obtain k objects that satisfy all the keywords defined in $q.t$ and therefore, the number of objects contained in *heap* may be less than k.

The algorithm begins by obtaining the position in the array *nodes* of the root of the sub-tree that is between *start* and *end* (line 1). In lines 3-26, the case of the internal nodes is solved using a depth-first search (DFS) traversal. Lines 4 and 5 retrieve the coordinate values corresponding to the direction of the partition according to the *depth* in the *i*KD-Tree, both for the point $q.l$ and for the point p of the node. In lines 6-15 the algorithm processes the case in which the point $q.l$ is to the left of the point p, since the value of c_q is less than or equal to that of c_p. Then, function *checkLS*(., ., ., .) (line 7) verifies if the bitmap *BR* associated to the left subspace of the node contains all the keywords indicated in $q.t$, that is, if $q.t$ is a subset of the union of the keywords of all the points located in the left subspace. If so, the traversal is continued by recursively accessing the first half (sub-tree) of the subarray of *nodes* between *start* and *end* (line 8). If it does not contain them, the other branch of the *i*KD-Tree is processed (lines 12-14). Then, function *existsRight*(., .) (explained below) verifies if it is necessary to explore the points of the subspace to the right of p with the goal of finding out if they may improve the best solution achieved so far. Analogously, lines 16-25 process the case when the path must continue on the right side of p.

Lines 27-29 process the case in which the depth traversal has reached a leaf node, which must be revised, or when the algorithm has returned from recursion and the solution must be improved. In such cases, the function *checkKeywords*(., .) (explained below) verifies if the point contains all the keywords specified by $q.t$. If so, the point is considered a candidate and it is used to update the *heap* using the function *updateCandidate*(., ., ., .).

The algorithm uses the following non-trivial functions:

- **checkKeywords(**$q.t$, *pos***)** verifies if the keywords in the query $q.t$ exist in the visited point *pos*.
- **checkLS(**$q.t$, *pos*, *start*, *end***)** and **checkRS(**$q.t$, *pos*, *start*, *end***)** verify if the keywords in the query $q.t$ exist in the left (conceptual bitmap *LS*), resp. right (conceptual bitmap *RS*), subspace of the visited point *pos*.
- **updateCandidate(***heap*, p, $q.l$, $q.k$**)** is used to add point q to the candidates *heap*. It is assumed that p is also a candidate, that is, it contains all the keywords of the query. The function decides wether to insert p to *heap* or not. To do this, it verifies if the size of the heap is less than k, if so, p is inserted. Otherwise, it is verified if the euclidean distance between p and $q.l$ is less than the distance between $q.l$ and the point in the root of *heap*. If so, the point on the root is removed and p is inserted.
- **existsLeft(***heap*, $q.l$**)** and **existRight(***heap*, $q.l$**)** are used to decide, when returning from recursion in the internal nodes, if it is necessary to continue the traversal through the subspace opposite to the one already accessed. If the size of the heap is less than k the traversal is carried out anyway. Otherwise, it is verified if the circumference centered on the point c of the root of the *heap* and radius equal to the distance between $q.l$ and c intersects the subspace not explored by the recursion.

Algorithm 1 searchBkSKQ(q, $start$, end, $depth$, $heap$)

1: $mid \leftarrow \frac{(start+end)}{2}$
2: $p \leftarrow nodes[mid]$
3: **if** $(start \neq end)$ {Internal node} **then**
4: $\quad c_q \leftarrow getCoordinate(depth, q.l)$
5: $\quad c_p \leftarrow getCoordinate(depth, p)$
6: $\quad$**if** $(c_q \leq c_p)$ {Search left} **then**
7: $\quad\quad$**if** $(checkLS(q.t, mid, start, end))$ **then**
8: $\quad\quad\quad searchBkSKQ(q, start, mid - 1, depth + 1, heap)$
9: $\quad\quad\quad$**if** $existsRight(heap, q.l) \wedge checkRS(q.t, mid, start, end)$ **then**
10: $\quad\quad\quad\quad searchBkSKQ(q, mid + 1, end, depth + 1, heap)$
11: $\quad\quad\quad$**end if**
12: $\quad\quad$**else if** $(checkRS(q.t, mid, start, end))$ **then**
13: $\quad\quad\quad searchBkSKQ(q, mid + 1, end, depth + 1, heap)$
14: $\quad\quad$**end if**
15: $\quad$**end if**
16: $\quad$**if** $(c_q \geq c_p)$ {Search right} **then**
17: $\quad\quad$**if** $(checkRS(q.t, mid, start, end))$ **then**
18: $\quad\quad\quad searchBkSKQ(q, mid + 1, end, depth + 1, heap$
19: $\quad\quad\quad$**if** $existsLeft(heap, q.l) \wedge checkLS(q.t, mid, start, end)$ **then**
20: $\quad\quad\quad\quad searchBkSKQ(q, start, mid - 1, depth + 1, heap)$
21: $\quad\quad\quad$**end if**
22: $\quad\quad$**else if** $(checkLS(queryKey, mid, start, end))$ **then**
23: $\quad\quad\quad searchBkSKQ(q, start, mid - 1, depth + 1, heap)$
24: $\quad\quad$**end if**
25: $\quad$**end if**
26: **end if**
27: **if** $checkKeywords(q.t, mid)$ **then**
28: $\quad updateCandidate(heap, p, q.l, q.k)$
29: **end if**

B. EVALUATION OF R*k*SKQ QUERIES

In this section, we describe an algorithm to compute the RkSKQ queries defined in Section II-A2. In a nutshell, the strategy consists of searching for the spatio-textual objects that are closer to the query point, ranking the results by the weighted sum (score) (1) of the spatial proximity (euclidean distance) and the textual relevance (number of keywords associated with the object).

The algorithm is based on branch and prune, and it is similar to classical algorithms for K nearest neighbors. The main difference is that we use the *score* instead of just the euclidean distance or any other spatial distance. When branching, the algorithm evaluates the score of each subspace by using conceptual bitmaps LS and RS, and the boundaries of the subspace associated with the visited node. Then, it proceeds to the node with highest score. The same score is also used to prune the traversal. If the score of a subspace does not improve the best known solution so far, the subspace is discarded. A particular case is when none of the searched keywords is presented in the subspace. In such case, the score is zero and the subspace is discarded.

An example of the RkSKQ algorithm can be seen in the conceptual tree of Fig. 7. For a better understanding of each

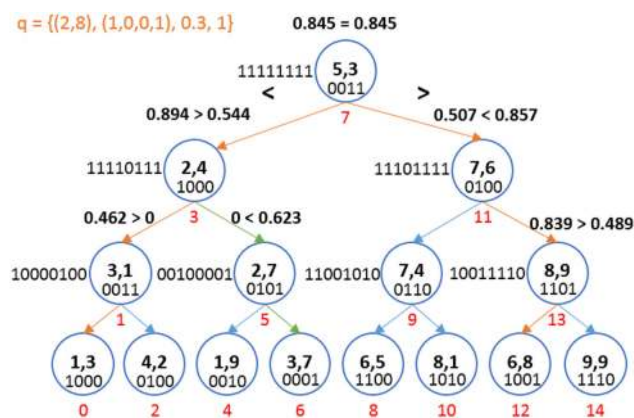

FIGURE 7. Example of a RkSKQ query showing the traversal over the conceptual tree of the c*Bi*K structure. Each node contains the spatial information, the keywords of the point, and the summary bitmap for an efficient pruning.

step, we provide Table 1. The figure shows a query q with four parameters: the query point (2, 8), the keywords that are represented as a bitmap, the weight $\alpha = 0.3$ that indicates that the total score is 30% due to spatial proximity and 70% due to textual relevance, and the value of $k = 1$.

TABLE 1. Step by step execution of the R*k*SKQ query in Fig. 7.

Iteration	i	Point	Score				
			Spatial	Textual	Total	LS	RS
1	7	(5,3)	0.484	0.5	0.495	0.845	0.845
2	3	(2,4)	0.646	0.5	0.544	0.894	0.544
3	1	(3,1)	0.374	0.5	0.462	0.462	0.000
4	0	(1,3)	0.549	0.5	0.515		
5	5	(2,7)	0.912	0.5	0.623	0.000	0.623
6	6	(3,7)	0.875	0.5	0.612		
7	11	(7,6)	0.523	0	0.000	0.507	0.857
8	13	(8,9)	0.462	1	0.839	0.839	0.489
9	12	(6,8)	0.646	1	0.894		

Note that the traversal path is determined by the classification score of the summaries. At iteration 4, the point $(1, 3)$ corresponds to a leaf node (i.e. it does not have a summary) so the process is completed and the point is considered as a candidate solution. Then, for each point on the recursion backtrack, the algorithm tries to improve the solution. Note that when backtracking to the point $(2, 4)$, the other subspace is revised since *RS* estimates that there may be better candidates having a score higher than the current best and, indeed, the best solution becomes the point $(2, 7)$.

Since the summary score of the root node is equal in both sides, the algorithm should keep looking for candidate points, in step 9, the final solution is reached at point $(6, 8)$ with a total score 0.894. Note that point $(2, 7)$ is the closest to the query point q, but as the textual component has been given greater importance ($\alpha = 0.3$), the final solution corresponds to a point a little farther away, but containing a better match of the searched keywords.

This procedure is synthesized in Algorithm 2, which receives five parameters: q, $start$, end, which were described at the beginning of the section, together with a variable h that corresponds to a Min-Heap of size k used to keep the candidate points ranked by score from lowest to highest, and the variable α that weights the importance of the spatial and textual scores.

The algorithm first obtains the median of *start* and *end* (line 1), corresponding to the position of the point to be evaluated in the array *nodes*. Then in lines 2-4, the *spatialS*, the *textualS*, and the *totalScore* of the accessed point with respect to the query point are computed.

Then, lines 5-23 correspond to the case when the accessed point is an internal node of the tree, and it is necessary to determine to which subspace the traversal should continue. Line 6 obtains the textual scores of the left ($textualScores_{LS}$) and right ($textualScores_{RS}$) summaries to, then, obtain the total scores ($score_{LS}$ and $score_{RS}$). The traversal uses this information to decide which subspace should visit next.

Lines 9-14 are related to the case in which there are more keyword matches in the left subspace than in the right subspace ($score_{LS} > score_{RS}$), hence the traversal process such subspace in line 10. In the backtrack of the recursion, function *goToR*(., ., .) determines if it is necessary to access the other subspace in order to improve the solution. If necessary, this is accessed on line 12. Analogously, lines 14-19 are related to the opposite case in which the right subspace has a greater score. In lines 19-22, the case when both subspaces have the same score is processed by recursively traversing both of them.

Finally, lines 24-26 are related to the case in which the traversal has either reached a point corresponding to a leaf node, or it is returning from the recursion and the current point has to be considered as a candidate to improve the solution.

The non-trivial functions used in Algorithm 2 are explained as follows:

- **summaryTextualScores(**$q.t$, pos, $start$, end**):** returns an object with two attributes, the textual score (3) of the left and right summary, i.e. $textualScores_{LS}$ and $textualScores_{RS}$, respectively.
- **updateCandidate(**$heap$, p, $totalScore$, $q.k$**):** is similar to the function used in Alg. 1, and it decides the insertion of p into the *heap* depending on its score. As *heap* is a Min-Heap, the points with higher scores will be kept in the heap.
- **goToL(**$heap$, $score_{LS}$, $q.k$**)** and **goToR(**$heap$, $score_{RS}$, $q.k$**):** Similar to functions *existsLeft*(., .) and *existsRight*(., .) used in Alg. 1. It decides, depending on the score of the received summary and that of the point in the root of the heap, if it is necessary to continue the traversal in the subspace opposite to the one already traversed (i.e. if there may be candidates with better scores in such subspace).

C. EVALUATION OF *b*RS-SKQ QUERIES

The general idea is to traverse down the tree until the specified range is located. The access to the corresponding child is done as long as the *RL* reports that all the searched keywords are present in the corresponding subspace, stopping the traversal, otherwise. The navigation ends when a leaf node is reached, then, in the return of the recursion, it is checked if the visited nodes are contained in that region and if they contain an exact match of the requested keywords, if so, the point is added to the result list.

Fig. 8 illustrates this procedure with a query q that has three parameters: the two endpoints of the spatial region in Fig. 8(a), and the searched keyword represented by its bitmap 0100. The traversal is shown in Fig. 8(b), starting from the root node and continuing until point $(2, 7)$ that intersects the query region. Hence, candidate points may exist in both subspaces and they must be visited. However, as *RL* indicates that the searched keyword is not present in any of the children, the traversal backtracks and checks if the visited points are contained in the query region. The final result only contains the point $(2, 7)$. Note that point $(3, 7)$ is contained in the range, but it does not contain the searched keyword.

Algorithm 3 shows the procedure of the *b*RS-SKQ. The function receives 4 parameters, q contains the query information with attributes $q.l_1$ and $q.l_2$ the query points and $q.t$ the searched keywords. Before solving the query, the coordinates

Algorithm 2 **searchRkSQK**$(q, start, end, h, \alpha)$

1: $mid \leftarrow \frac{(start+end)}{2}$
2: $spatialS \leftarrow \delta(nodes[mid].l, q.l)$ {see (2)}
3: $textualS \leftarrow \theta(nodes[mid].t, q.t)$ {see (3)}
4: $totalScore \leftarrow totalScore(spatialS, textualS, \alpha)$ {implements (1)}
5: **if** $(start \neq end)$ **then**
6: $\quad textualScores \leftarrow summaryTextualScores(q.t, mid, start, end)$
7: $\quad score_{LS} \leftarrow totalScore(spatialS, textualScores_{LS}, \alpha)$
8: $\quad score_{RS} \leftarrow totalScore(spatialS, textualScores_{RS}, \alpha)$
9: $\quad$ **if** $(score_{LS} > score_{RS})$ {Search left} **then**
10: $\quad\quad searchRkSQK(q, start, mid-1, h, \alpha)$
11: $\quad\quad$ **if** $goToR(h, score_{RS}, q.k)$ **then**
12: $\quad\quad\quad searchRkSQK(q, mid+1, end, h, \alpha)$
13: $\quad\quad$ **end if**
14: $\quad$ **else if** $(score_{RS} > score_{LS})$ {Search right} **then**
15: $\quad\quad searchRkSQK(q, mid+1, end, h, \alpha)$
16: $\quad\quad$ **if** $goToL(h, score_{LS}, q.k)$ **then**
17: $\quad\quad\quad searchRkSQK(q, start, mid-1, h, \alpha)$
18: $\quad\quad$ **end if**
19: $\quad$ **else if** $(score_{RS} \neq 0)$ {Same score but $\neq$ to zero, search both sides} **then**
20: $\quad\quad searchRkSQK(q, start, mid-1, h, \alpha)$
21: $\quad\quad searchRkSQK(q, mid+1, end, h, \alpha)$
22: $\quad$ **end if**
23: **end if**
24: **if** $(totalScore \neq 0)$ {At least one keyword found} **then**
25: $\quad updateCandidate(h, p, totalScore, q.k)$
26: **end if**

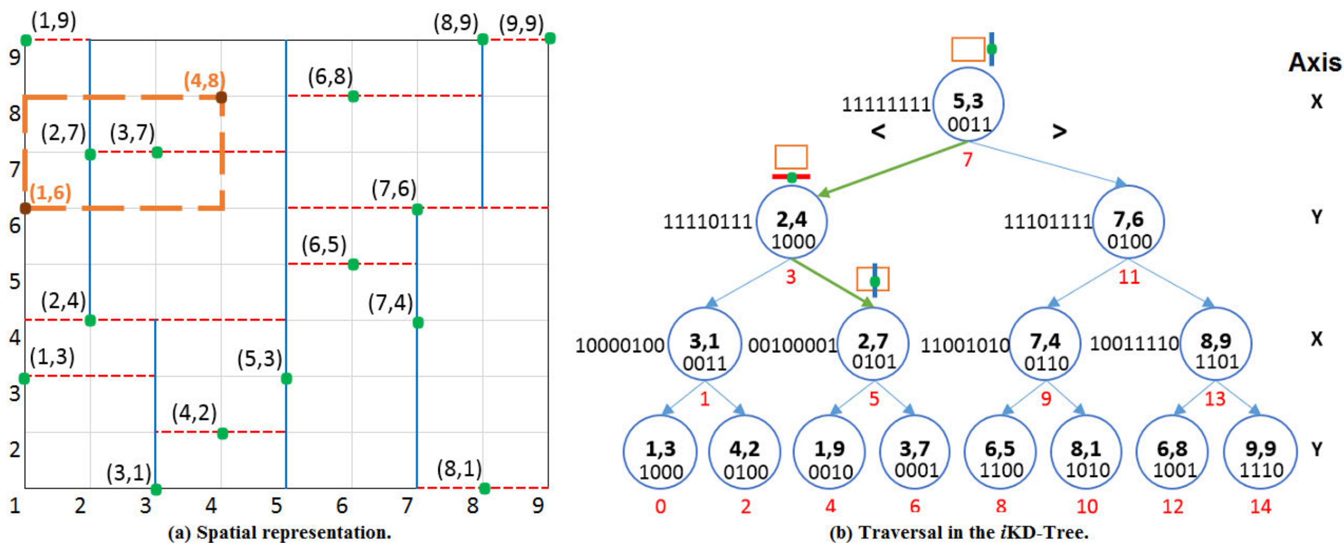

FIGURE 8. Example of *b*RS-SKQ query with $q = \{(1, 6), (4, 8), (0100)\}$.

are ordered so that $q.l_1$ is the lower left point and $q.l_2$ the upper right point of the determined region. The variables *start*, *end* and *depth* are used as in previous algorithms. Note that there is a *result* list as a global variable in which the objects that satisfy the requested spatial and textual conditions are stored.

The algorithm first obtains the median of the values of *start* and *end* (line 1), which corresponds to the position of the

Algorithm 3 rangeSearching(q, $start$, end, $depth$)

1: $mid \leftarrow \lfloor \frac{start+end}{2} \rfloor$
2: $p \leftarrow nodes[mid]$
3: **if** $(start \neq end)$ **then**
4: $c_M \leftarrow getCoordinate(depth, q.l_2)$ {Right upper coordinate}
5: $c_m \leftarrow getCoordinate(depth, q.l_1)$ {Lower left coordinate}
6: $c_p \leftarrow getCoordinate(depth, p)$
7: **if** $(c_M \leq c_p)$) {Search left} **then**
8: **if** $checkLS(q.t, mid)$ **then**
9: $rangeSearching(q, start, mid - 1, depth + 1)$
10: **end if**
11: **else if** $(c_m > c_p)$ {Search right} **then**
12: **if** $checkRS(q.t, mid)$ **then**
13: $rangeSearching(q, mid + 1, end, depth + 1)$
14: **end if**
15: **else**
16: **if** $checkLS(q.t, mid)$ **then**
17: $rangeSearching(q, start, mid - 1, depth + 1)$
18: **end if**
19: **if** $checkRS(q.t, mid)$ **then**
20: $rangeSearching(q, mid + 1, end, depth + 1)$
21: **end if**
22: **end if**
23: **end if**
24: **if** $containsPoint(q, p) \wedge checkKeywords(q.t, mid)$ **then**
25: $result.insert(p)$
26: **end if**

point accessed in the array *nodes*. Then, the point is retrieved and stored in variable p.

Lines 3-23 evaluate the case when the revised point is an internal node. First, in lines 4-6 the value of the coordinate according to the partition is revised, c_m and c_M correspond to the lower left and upper right corners of the region, and c_p is the actual value of the evaluated point. In order to determine to which subspace the search should continue, the following three cases are evaluated:

- **Case 1:** When $c_M \leq c_p$, the partition of the point is located to the right (or above) the query region (line 7), the search continues to the left subspace.
- **Case 2:** When $c_m > c_p$, the partition of the point is located to the left (or under) the query region (line 11), the search continues to the right subspace.
- **Case 3:** In other case (line 15), the partition of the point intersects the query region, hence, the search continues in both subspaces.

Note that in all cases, in order to make the recursive call, it is necessary to verify that the subspace contains the requested keywords using the functions $checkLS(.,.)$ and $checkRS(.,.)$, according to the case. If the subspace contains them, the traversal continues as there are candidate points within the range of queried positions.

Finally, lines 24-26 evaluate if the point is contained in the query region using function $containsPoint(q, p)$. Moreover, function $checkKeywords(q.t, mid)$ determines if the searched keywords completely match the keywords of the point. In such case, the point is added to the list *result*.

D. ANALYSIS OF THE TEMPORAL/SPATIAL COMPLEXITY OF THE ALGORITHMS

In this section we explain the temporal complexity of the algorithms to find the nearest neighbor (BkSQK and RkSKQ), and also the range searching (bRS-SKQ). In addition, the spatial complexity is analyzed to determine the amount of memory (RAM) required by the structure of cBiK.

For all algorithms, the construction of cBiK considering n nodes and m keywords. The worst case happens when all the n points have all the m existing keywords and, therefore, the memory used is represented by (4), which includes the substructures used in cBiK. It is important to emphasize that this is a worst case analysis and, as it can be seen in Section VII, in practice the space is much lower due to the effectiveness of the SD-array to encode sparse bitmaps as BK and BR:

$$Storage = S_{points} + S_{keys} + S_{summary} + S_{map} + S_{hashing} \quad (4)$$

- S_{points}: Corresponds to the totality of the spatial points (coordinates x and y) that are stored with variables of type *double* (8 bytes). Hence, the memory consumption

is given by (5).

$$S_{points} = n \cdot (16\ bytes) \quad (5)$$

- S_{keys}: It refers to the storage of Bitmap Keywords (BK), which represents the keywords associated with each point. As in the worst case all n points have m bits, then the memory used is shown in (6).

$$S_{points} = n \cdot \left(\frac{m}{8}\ bytes\right) \quad (6)$$

- $S_{summary}$: Equation (7) shows the bytes used by the Bitmap Summary (BR). As just the explicit summaries are stored, the nodes of the last two levels of the tree have to be omitted. Such nodes are represented as the following summation $\sum_{i=0}^{h-3} 2^i$, with h the height of the tree.

$$S_{summary} = \overbrace{\left(2^{\lceil \log_2 n \rceil - 2} - 1\right)}^{explicit\ RL's} \cdot 2 \cdot \left(\frac{m}{8}\ bytes\right) \quad (7)$$

- S_{map}: Represents the storage of the Bitmap Map (BM), which considers twice the number of nodes, as shown in (8).

$$S_{map} = 2 \cdot \left(\frac{n}{8}\ bytes\right) \quad (8)$$

- $S_{hashing}$: Equation (9) corresponds to the mapping hash-structure used to transform keywords into their corresponding numeric identifiers, being l_i (with $i > 0$) the length of the keyword.

$$S_{hashing} = \mathcal{O}\left(\sum_{i=1}^{m} size(l_i)\right) \quad (9)$$

where $size(l_i)$ is the length of string in bytes.

For all the above, the spatial complexity corresponds to $\mathcal{O}(n \cdot m)$, in the worst case.

Regarding the temporal complexity of the algorithms, we can review the following two cases:

1) THE *k* NEAREST NEIGHBORS (*k*NN)

If $k = 1$ and $k \ll n$, the worst case to find the nearest neighbor (1NN) happens if all the points contain all the searched keywords and, at the same time, they are located in a circle centered in the query point q. This layout forces the traversal of all the nodes of the tree as all the distances to the point q are equal. Hence, the temporal complexity is $\mathcal{O}(n)$.

On the other hand, if the points are uniformly distributed, the worst case to find the 1NN is $\mathcal{O}(\log n)$ on average.

If $k > 1$ and $k \ll n$, the operations in the *heap* to deliver the results have to be also considered, which has size at most k.

$$\overbrace{\mathcal{O}(n \cdot \log k)}^{Nodes\ processing} + \overbrace{\mathcal{O}(n \cdot |q.t|)}^{Bitmap\ processing}$$

Hence, the temporal complexity for the kNN in the worst case is shown in (10), being $|q.t|$ the cardinality of the searched keywords.

$$\mathcal{O}(n \cdot (\log k + |q.t|)) \quad (10)$$

2) RANGE SEARCHING

For this type of query, the worst case occurs when all the points inside the query region R, contain all the requested keywords. To obtain the temporal complexity of the balanced *i*KD-Tree, with height $h = \lceil \log_2 n \rceil$, we must determine which is the maximum number of subspaces $Q(n)$ intersected in the KD-Tree by cutting the plane with a line l. For this, the key idea is to consider two levels of the tree at the same time. If you consider first a vertical cut and then a horizontal one, 4 subspaces are obtained, each with $\frac{n}{4}$ points. Then, the line will intersect two subspaces and the remaining subspaces will be outside or completely within R. The recurrence that expresses the above is the following:

$$Q(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2Q(\frac{n}{4}) + 2 & \text{if } n > 1 \end{cases}$$

This recurrence is resolved to $Q(n) = \mathcal{O}(\sqrt{n})$. Also, the total time to report all the points P contained in the region Q contributes with $\mathcal{O}(P)$ time. Finally, the temporal complexity for the range query in the worst case is $\mathcal{O}(\sqrt{n}+P)$.

VII. EXPERIMENTAL EVALUATION

This section provides a comprehensive evaluation of *c*B*i*K that analyzes its space consumption and its performance to resolve the three different queries proposed in this paper: B*k*SKQ, R*k*SKQ, and *b*RS-SKQ. Our prototype[5] is coded in C++, and uses different bitmap implementations available in the SDSL [39] to build the corresponding bitmaps of *c*B*i*K.

We compare *c*B*i*K to S2I [9], one of the most competitive approaches in the state of the art [7] for the current scenario. S2I is a disk-oriented index, in contrast to cB*i*K, which resides on main-memory. Although this fact penalizes S2I, which always pays I/O costs, the following experimentation enables disk and memory-based solutions to be effectively compared. Note that I/O is one of the main bottlenecks of geo-textual indexes competitive approaches in the state of the art [7], and our approach tries to overcome it by using succinct data structures that are more likely to fit in main memory. The S2I prototype[6] is coded in Java.

a: DATASETS

We consider two different datasets[7] for our experiments, both used in [40]:

- The **POI's** dataset contains 1.1 million real-world points of interest (POI) obtained from the social network Foursquare. Each POI provides its geographical coordinates and a short description.
- The **Twitter** dataset is a collection of 20 million geo-tagged *tweets*, each one containing a short text and a geographical description of the location from which the tweet was published. It is worth noting that S2I is

[5]https://github.com/csanjuanc/cBiK
[6]The S2I prototype has been kindly provided by its author.
[7]Datasets available at http://www.ntu.edu.sg/home/gaocong/datacode.htm

TABLE 2. Dataset description.

Datasets	Objects	Words per object	Different words	Total words
POI's	1.1 million	4.00	261,212	4,389,929
Twitter1M	1.0 million	4.34	283,806	4,342,238
Twitter3M	3.0 million	4.28	585,624	12,848,921
Twitter5M	5.0 million	4.27	825,971	21,338,978
Twitter10M	10.0 million	4.70	1,364,787	47,028,156

not able to index the whole dataset, forcing us to work with smaller data. We divide the Twitter dataset into four smaller ones, namely Twitter1M, Twitter3M, Twitter5M and Twitter10M, that contain 1, 3, 5, and 10 million tweets.

Table 2 summarizes the most relevant features of all these datasets. Note that the average number of words per object is quite similar in all cases (between 4 and 4.7), so the number of total words is proportional to the dataset size. On the other hand, the number of different words also increases with the dataset size, ranging between $261,212$ and $1,364,787$, for POI's and Twitter10M, respectively. Note that the number of different words is an important metric for cB*i*K because it determines the size of the bitmaps.

b: QUERIES

We designed a testbed of randomly generated queries for each dataset in our setup. For B*k*SKQ and R*k*SKQ, we generated five sets of 1,000 queries, each one providing a point (x, y) and a list of k keywords ($1 \leq k \leq 5$). Latitude and longitude values were randomly generated at the $[-90, 90]$ and $[-180, 180]$ intervals, respectively, while the keywords were also randomly chosen from the collection of different words used in each dataset. This decision ensures that all queries return, at least, one result.

Queries for *b*RS-SKQ follow a similar pattern, but regions were queried instead of points. Each region is defined around an existing point in the dataset, which is randomly chosen. The corresponding region is the (squared) bounding box that has the selected point as its center.

We consider five different-size regions according to the length of the diagonal of the bounding box: *1km*, *2km*, *5km*, *10km*, and *20km*. The POI's dataset describes points that cover almost half of the Earth (note that the distance between its farthest points is $18,909.6$ kilometers), so regions of *1Km* diagonal are just 0.005% of the whole area of the dataset, while regions of *20Km* cover 0.106% of this area. Twitter datasets cover smaller areas, so the query regions are proportionally larger in these cases.

The performance of our solution is benchmarked in the following sections. Note that all experiments were run on an Intel® Core™ *i*5@3.4GHz (4 Cores), 8GB of RAM, and a 1TB SATA disk, over Ubuntu 16.04 LTS (64 bits).

TABLE 3. Dataset construction time in seconds.

Datasets	S2I	cBiK
POI's	21,356	22
Twitter1M	15,424	24
Twitter3M	224,756	78
Twitter5M	604,011	134
Twitter10M	N/A	1,347

TABLE 4. Storage usage.

Dataset	Original (MB)	S2I (MB)	c*Bi*K (MB) Total	Hashing	Points	BM	BK	BR
POI's	81.9	246	**96.7**	3.09	36.2	0.3	8.6	48.5
Twitter1M	62.8	281	**100.9**	3.45	24.6	0.2	9.4	63.2
Twitter3M	191.0	845	**319.8**	7.58	73.9	0.7	29.6	208.0
Twitter5M	320.0	1,403	**561.1**	10.90	123.0	1.2	50.0	376.0
Twitter10M	668.0	N/A	**1185.5**	18.09	246.0	2.38	114.0	805.0

A. CONSTRUCTION TIME AND STORAGE USAGE

Table 3 shows construction time in seconds of both the baseline, S2I, and our proposal, c*Bi*K, for all the datasets used in the experiments. From these results, we can conclude that our proposal clearly outperforms the baseline in several orders of magnitude. Taking as an example the dataset Twitter5M, S2I requires about a week to be built, whereas our proposal can be built in a couple of minutes. For the largest dataset, we were not able to build the baseline as the process exceeded the maximum secondary memory available (1TB) after three weeks of computation. Even for this dataset, our proposal can be built in about 20 minutes. This is an important result for the scalability of our proposal.

The space complexity measures the amount of storage space used by each index. As showed in Table 4, S2I uses more space than c*Bi*K to index each dataset (2.61 times more space, on average). In quantitative terms, c*Bi*K saves 841.9 MB to index Twitter5M with respect to S2I, and this improvement remains proportional for the other datasets.

Although space savings are expected due to the use of succinct data structures in c*Bi*K, this improvement is noticeable in practical terms because it enables larger datasets to be managed in main memory, avoiding costly I/O operations. Thus, managing smaller indexes also improves query performance, as shown below.

An interesting result regarding our structure is that most of the space is used by bitmap BR. Hence, a simple technique to reduce the space even further is not to store these bitmaps in all the levels but every x levels. This technique obviously impacts in the time performance, so parameter x would provide a time-space trade-off. To further explore this idea is proposed as an open problem.

B. QUERY PERFORMANCE

This section presents and discusses performance figures for the three different queries considered in our setup: B*k*SKQ, R*k*SKQ, and *b*RS-SKQ. In all cases, averaged query times

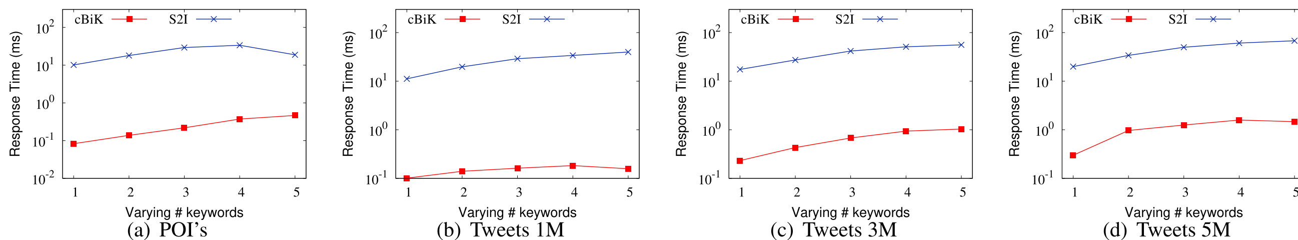

FIGURE 9. B*k*SKQ **performance according to the number of keywords provided by the query (*top* − 5 queries).**

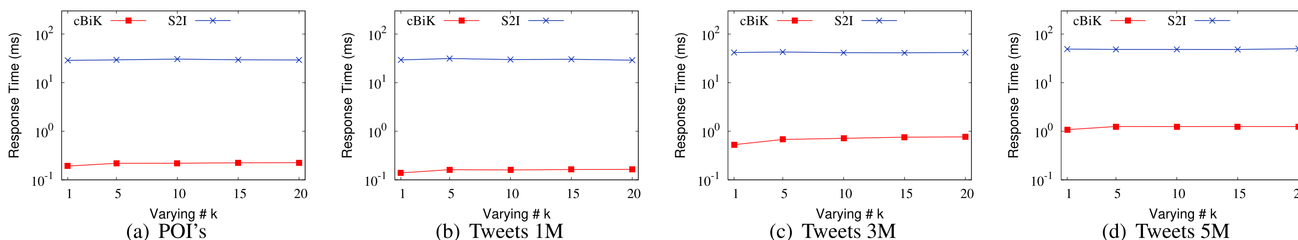

FIGURE 10. B*k*SKQ **performance according to the number of requested results (all queries provide 3 different keywords).**

are reported; i.e. the corresponding query set (of $1,000$ queries) is executed, and the total running time is divided by $1,000$. All times are reported in milliseconds per query.

1) BOOLEAN TOP-*k* SPATIAL KEYWORD QUERY

BkSKQ is a boolean query that retrieves the *Top-K* closest locations (to the query point) that match all required keywords. Thus, the performance of this query is affected by *the number of requested results*, and *the number of keywords* provided by the query.

Fig. 9 reports query times as a function of the number of keywords provided by the query: from 1 to 5 keywords. Note that all of these queries ask for the corresponding *top-5*. cBiK is more than 1 order of magnitude faster than S2I, but the improvement is almost of 2 orders of magnitude for the smallest datasets: POI's and Twitter1M. S2I reports query times between 10 and 68ms per query, and cBiK $\approx 0.08 - 1.5$ms per query. Note that query times increase with the number of keywords because more comparisons must be done to evaluate each candidate. However, S2I performs better for 5 than for 4 keywords, in the POI's dataset. This is because many objects are described with few keywords (note that each point in POI's has 4 keywords on average, and the S2I prune algorithm is able to exploit this fact in this particular case.

Varying k barely affects query times, as showed in Fig. 10. In this case, all queries provide 3 different keywords and ask for the best 1, 5, 10, 15, and 20 results. Note that query times remain stable for all datasets and all *top-k* queries. As in the previous case, cBiK is more than one order of magnitude faster than S2I, but the difference is larger for the smallest datasets.

2) RANKED TOP-*k* SPATIAL KEYWORD QUERY

RkSKQ performance is also measured according to the number of keywords and the requested results, but it also considers the value of α.

We fixed $k = 5$ and $\alpha = 0.3$ to measure the effect of the number of keywords (we consider queries including from 1 to 5 keywords). It means that the score algorithm weights text relevance with 0.7 and spatial proximity with 0.3, in the corresponding Top-5 queries. Fig. 11 reports query times for this experiment. Although cBiK is still faster than S2I, the difference decreases with the number of keywords. S2I reports similar numbers than for BkSKQ, but cBiK pays an important overhead due to its pruning algorithm, which is less effective when not all the requested keywords must be present in the point description, and more candidates must be checked to obtain the best k results.

The experiment varying the number of k requested results draws similar conclusions than for BkSKQ. As showed in Fig. 12, query times remain quite stable from $k = 5$ to $k = 20$, although cBiK reports a small increase from $k = 1$ to $k = 5$. In any case, cBiK is 1 order of magnitude faster for all datasets, reporting times of $\approx 0.5 - 6.2$ miliseconds per query for all datasets.

Finally, the effect of α is evaluated. This experiment measures the impact of combining spatial and text filters in the query. We consider five different values for $\alpha = \{0.1, 0.3, 0.5, 0.7, 0.9\}$ to analyze whether query performance varies when one of the components is more relevant than the other. The case of $\alpha = 0.5$ is particularly interesting because it weighs similarly both the spatial proximity and the text relevance. All these queries ask for the corresponding *Top-5* and provide three different keywords.

Fig. 13 shows that cBiK and S2I report very stable numbers for all datasets. Thus, we can conclude that α barely affects

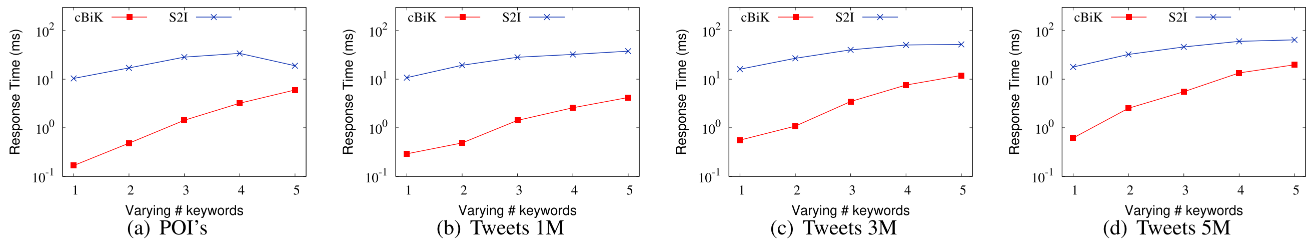

FIGURE 11. **R*k*SKQ performance according to the number of keywords provided by the query (*top* − 5 queries, $\alpha = 0.3$).**

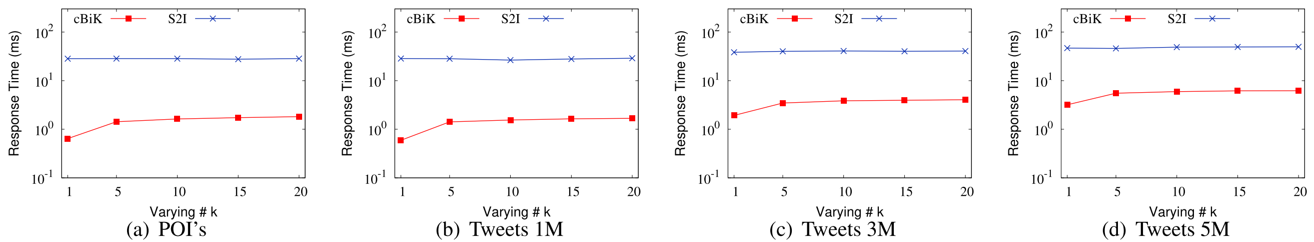

FIGURE 12. **R*k*SKQ performance according to the number of requested results (3 different keywords per query and $\alpha = 0.3$).**

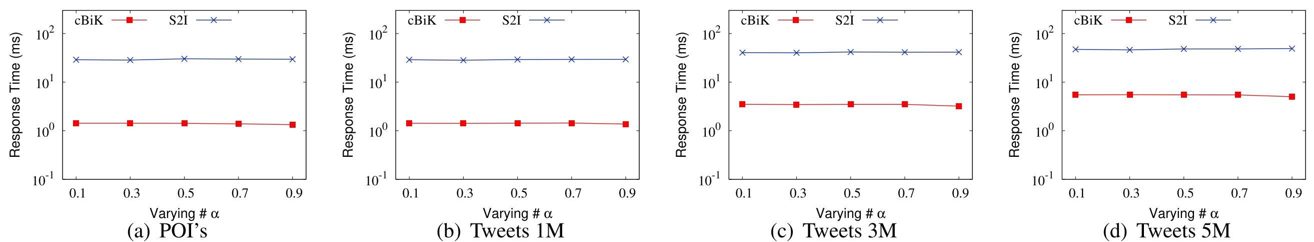

FIGURE 13. **R*k*SKQ performance varying α (Top-5 queries providing 3 different keywords).**

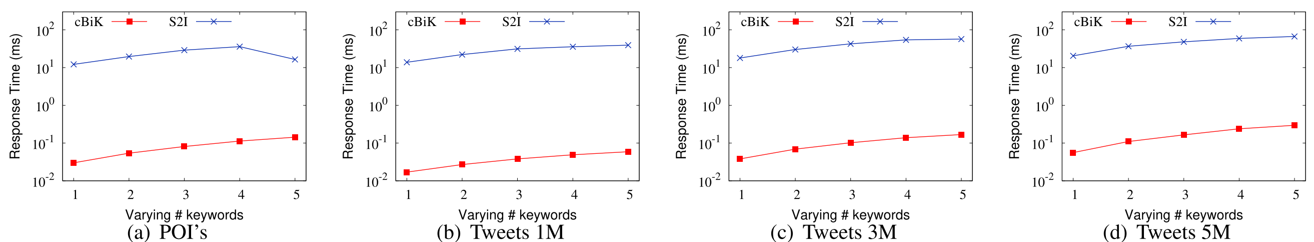

FIGURE 14. ***b*RS-SKQ performance according to the number of keywords provided by the query (all queries ask for regions of 10 Km).**

query performance, and *c*B*i*K is again one order of magnitude faster than S2I, reporting times from 1.3 to 5.5 milliseconds per query, while S2I performs at the level of dozens of milliseconds per query.

3) BOOLEAN RANGE SEARCHING SPATIAL KEYWORD QUERY

Finally, the *b*RS-SKQ performance is analyzed. In this case, we consider two parameters: the number of keywords and the size of the requested region.

Fig. 14 reports query times in function of the number of keywords. In this case, we fix query regions of 10Km and evaluate queries providing from 1 to 5 keywords. S2I reports similar figures than for B*k*SKQ (see Fig. 9), so it behaves similar when a reference point or a region are queried. On the other hand, *c*B*i*K reports competitive numbers from $\approx 0.017 - 0.29$ miliseconds per query. Note that its performance runs in parallel with S2I one, but it is two orders of magnitude faster, in all cases. It shows that *c*B*i*K is very efficient to resolve range-based queries.

Fig. 15 analyzes the effect of querying by different-length regions (1, 2, 5, 10, and 20Km of diagonal length), fixing 3 different keywords per query. Query times remain stable for all region lengths and all datasets, both for S2I and *c*B*i*K. Assuming that the longer the region, the greater the

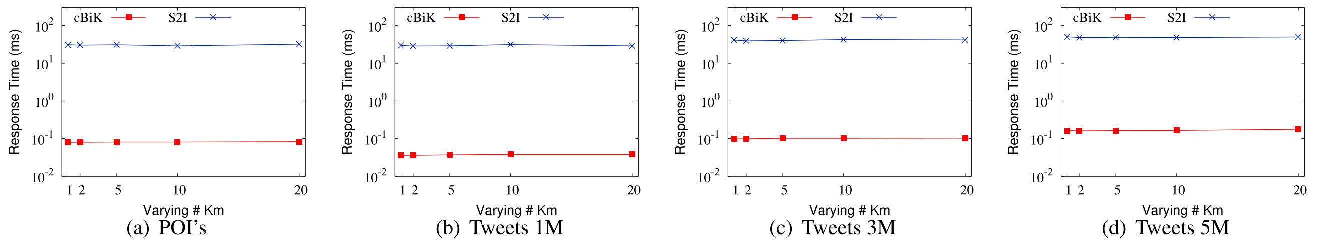

FIGURE 15. *b*RS-SKQ performance according to the length of the queried region (all queries provide 3 different keywords).

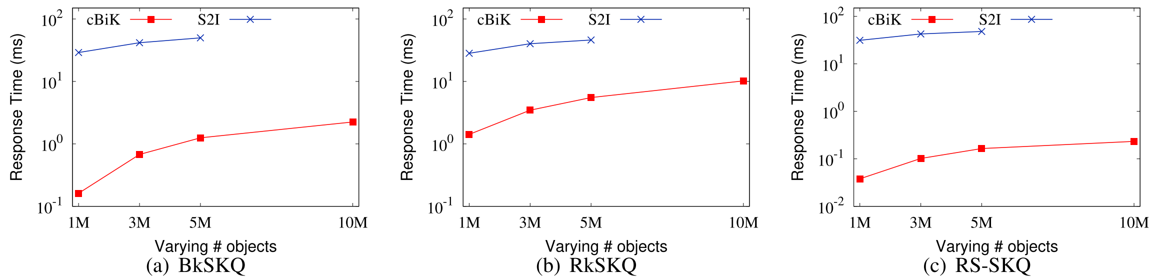

FIGURE 16. Scalability for queries BkSKQ with $key = 3$ and $k = 5$, RkSKQ ($\alpha = 0.3$) and RS-SKQ with $key = 3$ and $d = 10Km$.

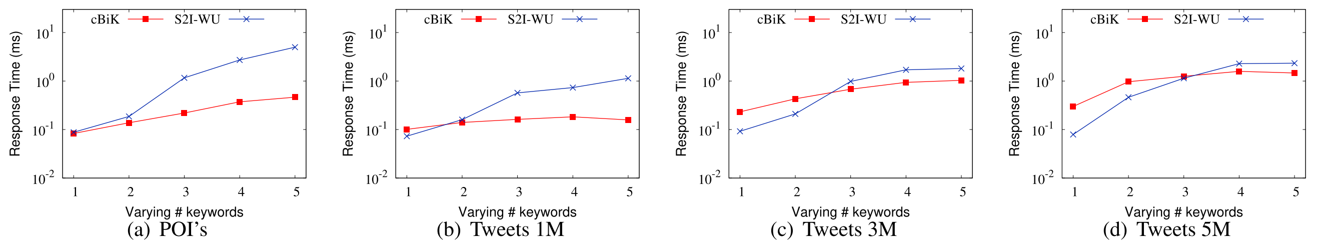

FIGURE 17. BkSKQ performance varying keywords (Top-5 queries).

number of points matching the query, this result means that *b*RS-SKQ performance does not depend on the number of retrieved points. *c*B*i*K also outperforms S2I in this scenario, but the improvement is higher, reaching up to 3 orders of magnitude for Tweets 1M.

Finally, Fig. 16 shows the scalability of the data structures with respect to the set size, defined as the number of spatio-textual objects. Both data structures present a linear behaviour with respect to the set size. However, the *c*B*i*K keeps it advantage of one to three orders of magnitude. The curve for S2I is not shown for the 10M set, because it was not possible to build such index with the available resources.

C. PLACING THE S2I IN MAIN MEMORY

In this section we show complementary experiments that compare our solution with the S2I when both solutions run in main memory. To do that, we use a *Warm-up* with the S2I. The idea of this technique is to run each query at least twice and just report the time of the second execution, which ensures that the part of the data structure involved in the running query is already in main memory, hence avoiding I/O operations. Although this kind of comparison is not completely fair (because it also favours cache-behaviour, making the data structure even faster than a main memory resident one), it provides a roughly estimation of the performance of the structure in main memory.

The results of these experiments are shown in Fig. 17, 18 and 19. In these figures, we label the S2I that uses the Warm-up technique as S2I-WU. First, it is important to observe the impact of the warm up technique in the S2I. As it can be observed, this technique reduces query times about one or two orders of magnitude, depending on the type of query. Second, Fig. 17 and 19 show that, even in this scenario, *c*B*i*K is faster than S2I-WU for the queries of type BkSKQ and bRS-SKQ. On the other hand, Fig. 18 shows that S2I-WU is faster than *c*B*i*k for R*k*SKQ queries. This is not a surprising result because: i) S2I is a data structure designed ad-hoc to efficiently solve this specific type of query and ii) it is a well known result in compact data structures that they are usually slower than classical data structures when running in the same level of the memory hierarchy. Overall, we can conclude that *c*B*i*K is a scalable solution to efficiently solve the three types of queries studiend in this work.

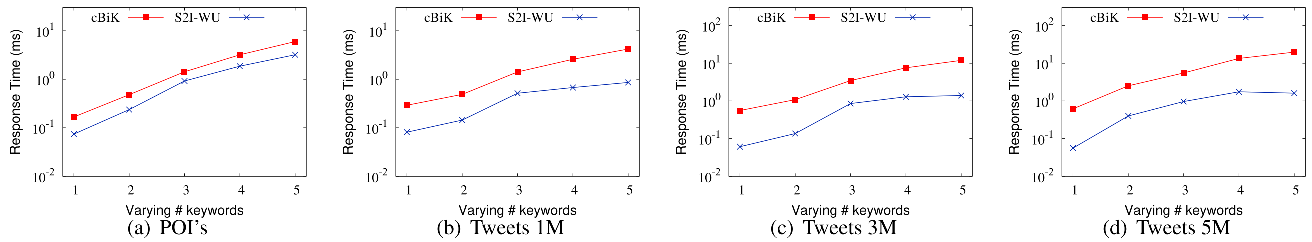

FIGURE 18. **R*k*SKQ performance varying keywords (Top-5 queries and alpha 0.3).**

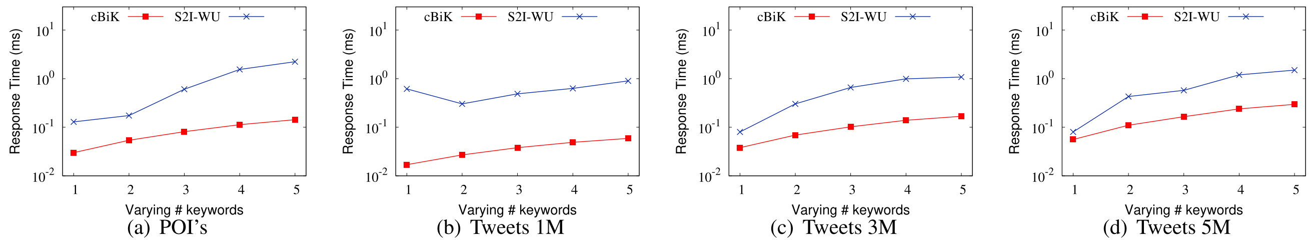

FIGURE 19. **bRS-SKQ performance varying keywords (10 Km).**

VIII. CONCLUSION AND FUTURE WORK

This paper proposes *c*B*i*K, the first compact data structure that manages geo-tagged datasets and allows spatial keyword queries to be resolved in main memory. Although this type of indexes performs more computations than traditional disk-based ones, they are more efficient by avoiding costly I/O operations.

Our experimentation verifies this fact using a selected testbed of real-world datasets, including points of interest descriptions and geo-located microposts from Twitter. *c*B*i*K is able to compact these datasets up to $35-40\%$ of the space used by a state-of-the-art index (S2I), while solving the three types of SKQs up to two orders of magnitude faster than S2I. When warming-up the S2I, to simulate another main-memory resident solution, both approaches are comparable in query times. Regarding construction time, our approach also scales much better with the number of objects to index. These numbers endorse our approach, and consequently an emergent line of research focused on the use of compact data structures for managing and querying big geo-tagged datasets.

We plan to enhance *c*B*i*K to support additional search capabilities, as part of our future work. On the one hand, we can replace the current mapping, which transforms keywords into integer identifiers, by a powerful compressed string dictionary that allows *inexact text queries*. More concretely, we plan to use the compressed FM-index dictionary [41] because it can be tuned to perform approximate string matching, with different string similarity measures [42], on the Burrows-Wheeler transform [43]. On the other hand, another interesting line of work is to provide semantic spatial keyword queries in *c*B*i*K. Based on the experiences of Tekli *et al.* [44], [45], we can add a compressed semantic index [46] that allows efficient semantic relationships between keywords to be efficiently navigated. In both types of searches, the additional indexes will be first queried to obtain the corresponding set of queries that are evaluated from the spatial perspective.

From a more applied perspective, the application of this approach in low memory devices, such as smart-phones, is promising. In such scenario, not just the space usage, but also the battery consumption must be reduced.

ACKNOWLEDGMENT

The authors would like to thank José R. Paramá (University of A Coruña, Spain) for his helpful advice and the original ideas that motivated this article.

REFERENCES

[1] A. Tabarcea, N. Gali, and P. Fränti, "Framework for location-aware search engine," *J. Location Based Services*, vol. 11, no. 1, pp. 50–74, Jan. 2017.

[2] G. Navarro, *Compact Data Structures—A practical approach*. Cambridge, U.K.: Cambridge Univ. Press, 2016.

[3] G. de Bernardo, "New data structures and algorithms for the efficient management of large spatial datasets," Ph.D. dissertation, Dept. de Computación, Univ. da Coruña, Coruña, Spain, 2014.

[4] F. Claude, "Space-efficient data structures for information retrieval," Ph.D. dissertation, Dept. de Computación, Univ. da Coruña, Coruña, Spain, 2013.

[5] J. C. Carlos San, R. G. Gutierrez, and M. A. Martinez-Prieto, "A compact memory-based index for spatial keyword query resolution," in *Proc. 37th Int. Conf. Chilean Comput. Sci. Soc. (SCCC)*, Nov. 2018, pp. 1–8.

[6] X. Cao, L. Chen, G. Cong, C. S. Jensen, Q. Qu, A. Skovsgaard, D. Wu, and M. L. Yiu, "Spatial keyword querying," in *Proc. 31st Int. Conf. Conceptual Model. (ER)*, 2012, pp. 16–29.

[7] L. Chen, G. Cong, C. S. Jensen, and D. Wu, "Spatial keyword query processing: An experimental evaluation," *Proc. VLDB Endowment*, vol. 6, no. 3, pp. 217–228, 2013. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2535569.2448955

[8] G. Cong, C. S. Jensen, and D. Wu, "Efficient retrieval of the top-k most relevant spatial Web objects," *Proc. VLDB Endowment*, vol. 2, no. 1, pp. 337–348, Aug. 2009. [Online]. Available: http://portal.acm.org/citation.cfm?id=1687666

[9] J. B. Rocha-Junior, O. Gkorgkas, S. Jonassen, and K. Nørvåg, "Efficient processing of top-k spatial keyword queries," in *Advances in Spatial and Temporal Databases* (Lecture Notes in Computer Science), vol. 6849. Berlin, Germany: Springer, 2011, pp. 205–222, doi: 10.1007/978-3-642-22922-0_13.

[10] M. Christoforaki, J. He, C. Dimopoulos, A. Markowetz, and T. Suel, "Text vs. Space: Efficient Geo-search query processing," in *Proc. 20th ACM Int. Conf. Inf. Knowl. Manage.*, 2011, pp. 423–432.

[11] A. Guttman, "R-trees: A dynamic index structure for spatial searching," *ACM SIGMOD Rec.*, vol. 14, no. 2, p. 47, Jun. 1984. [Online]. Available: http://portal.acm.org/citation.cfm?doid=971697.602266

[12] N. Roussopoulos, S. Kelley, and F. Vincent, "Nearest neighbor queries," *ACM SIGMOD Rec.*, vol. 24, no. 2, pp. 71–79, May 1995, doi: 10.1145/568271.223794.

[13] A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos, "Closest pair queries in spatial databases," *ACM SIGMOD Rec.*, vol. 29, no. 2, pp. 189–200, Jun. 2000, doi: 10.1145/335191.335414.

[14] S. Nutanong, E. H. Jacox, and H. Samet, "An incremental hausdorff distance calculation algorithm," *Proc. VLDB Endowment*, vol. 4, no. 8, pp. 506–517, May 2011, doi: 10.14778/2002974.2002978.

[15] H. Samet, "The quadtree and related hierarchical data structures," *ACM Comput. Surv.*, vol. 16, no. 2, pp. 187–260, Jun. 1984. [Online]. Available: http://portal.acm.org/citation.cfm?doid=356924.356930

[16] V. Gaede and O. Günther, "Multidimensional access methods," *ACM Comput. Surv.*, vol. 30, no. 2, pp. 170–231, Jun. 1998. [Online]. Available: http://portal.acm.org/citation.cfm?doid=280277.280279

[17] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, no. 9, pp. 509–517, Sep. 1975. [Online]. Available: http://portal.acm.org/citation.cfm?doid=361002.361007

[18] J. Zobel and A. Moffat, "Inverted files for text search engines," *ACM Comput. Surv.*, vol. 38, no. 2, p. 6, Jul. 2006. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1132956.1132959

[19] C. Faloutsos and S. Christodoulakis, "Signature files: An access method for documents and its analytical performance evaluation," *ACM Trans. Inf. Syst.*, vol. 2, no. 4, pp. 267–288, Oct. 1984.

[20] Y. Chen and Y. Shi, "Signature files and signature file construction," in *Encyclopedia of Database Technologies and Applications*. Hershey, PA, USA: IGI Global, Jan. 2005, pp. 638–645. [Online]. Available: http://services.igi-global.com/resolvedoi/resolve.aspx?doi=10.4018/978-%1-59140-560-3.ch105

[21] 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.

[22] D. Okanohara and K. Sadakane, "Practical entropy-compressed rank/select dictionary," Apr. 2006, *arXiv:cs/0610001*. [Online]. Available: http://arxiv.org/abs/cs/0610001

[23] S. Vaid, C. B. Jones, H. Joho, and M. Sanderson, "Spatio-textual indexing for geographical search on the Web," in *Advances in Spatial and Temporal Databases*, vol. 3633. Berlin, Germany: Springer, 2005, pp. 218–235. [Online]. Available: http://orca.cf.ac.uk/1840/

[24] Y. Zhou, X. Xie, C. Wang, Y. Gong, and W.-Y. Ma, "Hybrid index structures for location-based Web search," in *Proc. 14th ACM Int. Conf. Inf. Knowl. Manage. CIKM*, 2005, p. 155. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1099554.1099584

[25] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The R*-tree: An efficient and robust access method for points and rectangles," *ACM SIGMOD Rec.*, vol. 19, no. 2, pp. 322–331, May 1990.

[26] R. Hariharan, B. Hore, C. Li, and S. Mehrotra, "Processing spatial-keyword (SK) queries in geographic information retrieval (GIR) systems," in *Proc. 19th Int. Conf. Sci. Stat. Database Manage. (SSDBM)*, Jul. 2007, p. 16.

[27] I. De Felipe, V. Hristidis, and N. Rishe, "Keyword search on spatial databases," in *Proc. IEEE 24th Int. Conf. Data Eng.*, Apr. 2008, pp. 656–665. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4497474

[28] Z. Li, K. C. K. Lee, B. Zheng, W.-C. Lee, D. Lee, and X. Wang, "IR-tree: An efficient index for geographic document search," *IEEE Trans. Knowl. Data Eng.*, vol. 23, no. 4, pp. 585–599, Apr. 2011.

[29] D. Wu, G. Cong, and C. S. Jensen, "A framework for efficient spatial Web object retrieval," *VLDB J.*, vol. 21, no. 6, pp. 797–822, Dec. 2012.

[30] D. Wu, M. L. Yiu, G. Cong, and C. S. Jensen, "Joint top-K spatial keyword query processing," *IEEE Trans. Knowl. Data Eng.*, vol. 24, no. 10, pp. 1889–1903, Oct. 2012.

[31] J. B. Rocha-junior and K. Nørvåg, "Top-k spatial keyword queries on road networks," in *Proc. 15th Int. Conf. Extending Database Technol.*, 2012, pp. 168–179.

[32] D. Zhang, K.-L. Tan, and A. K. H. Tung, "Scalable top-k spatial keyword search," in *Proc. 16th Int. Conf. Extending Database Technol. EDBT*, 2013, p. 359. [Online]. Available: http://www.scopus.com/inward/record.url?eid=2-s2.0-84876799315&partn%erID=tZOtx3y1

[33] C. Zhang, Y. Zhang, W. Zhang, and X. Lin, "Inverted linear quadtree: Efficient top k spatial keyword search," *IEEE Trans. Knowl. Data Eng.*, vol. 28, no. 7, pp. 1706–1721, Jul. 2016.

[34] H.-J. Hong, G.-M. Chiu, and W.-Y. Tsai, "A single quadtree-based algorithm for top-k spatial keyword query," *Pervas. Mobile Comput.*, vol. 42, pp. 93–107, Dec. 2017, doi: 10.1016/j.pmcj.2017.09.009.

[35] R. Göbel, A. Henrich, R. Niemann, and D. Blank, "A hybrid index structure for geo-textual searches," in *Proc. 18th ACM Conf. Inf. Knowl. Manage. CIKM*, 2009, pp. 1625–1628. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1645953.1646188

[36] A. Khodaei, C. Shahabi, and C. Li, "Hybrid indexing and seamless ranking of spatial and textual features of Web documents," in *Database and Expert Systems Applications* (Lecture Notes in Computer Science), vol. 6261. Berlin, Germany: Springer, 2010, pp. 450–466.

[37] A. Almaslukh and A. Magdy, "Evaluating spatial-keyword queries on streaming data," in *Proc. 26th ACM SIGSPATIAL Int. Conf. Adv. Geographic Inf. Syst.*, Nov. 2018, pp. 209–218.

[38] R. A. Brown, "Building a balanced k-d tree in O(kn log n) time," 2014, *arXiv:1410.5420*. [Online]. Available: http://arxiv.org/abs/1410.5420

[39] S. Gog, T. Beller, A. Moffat, and M. Petri, "From theory to practice: Plug and play with succinct data structures," in *Experimental Algorithms* (Lecture Notes in Computer Science), vol. 8504. Cham, Switzerland: Springer, 2014, pp. 326–337.

[40] L. Chen, G. Cong, X. Cao, and K. L. Tan, "Temporal spatial-keyword top-k publish/subscribe," in *Proc. Int. Conf. Data Eng.*, Apr. 2015, pp. 255–266.

[41] M. A. Martínez-Prieto, N. Brisaboa, R. Cánovas, F. Claude, and G. Navarro, "Practical compressed string dictionaries," *Inf. Syst.*, vol. 56, pp. 73–108, Mar. 2016.

[42] N. Gali, R. Mariescu-Istodor, D. Hostettler, and P. Fränti, "Framework for syntactic string similarity measures," *Expert Syst. Appl.*, vol. 129, pp. 169–185, Sep. 2019.

[43] N. Zhang, A. Mukherjee, D. Adjeroh, and T. Bell, "Approximate pattern matching using the burrows-wheeler transform," in *Proc. Data Compress. Conf. DCC*, 2003, p. 458.

[44] J. Tekli, R. Chbeir, A. J. M. Traina, C. Traina, K. Yetongnon, C. R. Ibanez, M. Al Assad, and C. Kallas, "Full-fledged semantic indexing and querying model designed for seamless integration in legacy RDBMS," *Data Knowl. Eng.*, vol. 117, pp. 133–173, Sep. 2018.

[45] J. Tekli, R. Chbeir, A. J. M. Traina, and C. Traina, "SemIndex+: A semantic indexing scheme for structured, unstructured, and partly structured data," *Knowl.-Based Syst.*, vol. 164, pp. 378–403, Jan. 2019.

[46] M. A. Martínez-Prieto, M. A. Gallego, and J. D. Fernández, "Exchange and consumption of huge RDF data," in *The Semantic Web, Research and Applications* (Lecture Notes in Computer Science), vol. 7295. Berlin, Germany: Springer, 2012, pp. 437–452.

CARLOS E. SANJUAN-CONTRERAS received the bachelor's degree in informatics and the master's degree in computer science from the University of Bío-Bío, Chillán, Chile, in 2014 and 2019, respectively. His research interests include spatio-textual databases and compact data structures and algorithms.

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

MIGUEL A. MARTÍNEZ-PRIETO received the Ph.D. degree in computer science from the University of Valladolid (UVa), in 2010. He held a Postdoctoral position with the University of Chile, from 2010 to 2012. His scientific experience extends over the last 13 years. He is currently an Associate Professor with the Department of Computer Science, UVa. His main research interests include data engineering challenges, more concretely data compression and indexing, semantic web, and big data.

DIEGO SECO received the M.Sc. and Ph.D. degrees in computer science from the University of A Coruña, in 2006 and 2009, respectively. He is currently an Associate Professor with the Department of Informatics Engineering and Computer Science, Faculty of Engineering, Universidad de Concepción, Chile. He also participates with the Millennium Institute for Foundational Research on Data. His research interests include geographic information retrieval, geographic information systems, compressed data structures and algorithms for textual and geographic data, and bioinformatics.