Skia: Scalable and Efficient In-Memory Analytics for Big Spatial-Textual Data

Yang Xu, Bin Yao, Zhi-Jie Wang, Xiaofeng Gao, Jiong Xie, and Minyi Guo, *Fellow, IEEE*

Abstract—In recent years, spatial-keyword queries have attracted much attention with the fast development of location-based services. However, current spatial-keyword techniques are disk-based, which cannot fulfill the requirements of high throughput and low response time. With the surging data size, people tend to process data in distributed in-memory environments to achieve low latency. In this paper, we present the distributed solution, i.e., Skia (Spatial-Keyword In-memory Analytics), to provide a scalable backend for spatial-textual analytics. Skia introduces a two-level index framework for big spatial-textual data including: (1) efficient and scalable global index, which prunes the candidate partitions a lot while achieving small space budget; and (2) four novel local indexes, that further support low latency services for exact and approximate spatial-keyword queries. Skia can support common spatial-keyword queries via traditional SQL programming interfaces. The experiments conducted on large-scale real datasets have demonstrated the promising performance of the proposed indexes and our distributed solution.

Index Terms—Distributed systems, indexing, spatial-textual analysis.

1 Introduction

With the fast development of mobile phones and the location-based services (LBS), a tremendous amount of spatial-textual data has been generated, which is essentially a set of geo-tagged text segments. Moreover, there are increasing occasions, e.g., finding nearby places with specific keywords, where we need to perform analytics on spatial-textual data in both spatial and textual dimensions [1]. The requirement of such analytics has derived various spatial-keyword queries. Generally, a spatial-keyword query consists of a spatial predicate and complex keyword predicates, whose goal is to find places near to the location with similar text descriptions. Particularly, we aim to solve four types of spatial-keyword queries that are receiving particular attention [2], and the detailed examples are given below.

- **Exact Boolean range query:** Retrieve the places whose text descriptions exactly contain the keywords "relish" and "coffee" and whose positions are within 5km of the query position.
- **Exact Boolean kNN query:** Retrieve k places nearest to the query position whose text descriptions exactly contain the keywords "relish" and "coffee".
- **Approximate Boolean range query:** Retrieve the places whose text descriptions contain keywords similar to "relish" and "cake" (e.g., "rakish" and "bake") and whose positions are within 5km of the query position.
- **Approximate Boolean kNN query:** Retrieve k places nearest to the query position whose text descriptions contain keywords similar to "relish" and "cake".

- *Y. Xu, B. Yao , X.Gao and M. Guo are with the Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai, China 200240. Email: {xuyangit@, yaobin@cs., gao-xf@cs., guomy@cs.}sjtu.edu.cn.*
- *Z.-J. Wang is with the Guangdong Key Laboratory of Big Data Analysis and Processing, School of Data and Computer Science, Sun Yat-Sen University, Guangzhou, China. Email: wangzhij5@mail.sysu.edu.cn.*
- *Jiong Xie is a senior technical expert in the spatio-temporal cloud computing team of Alibaba Cloud. Email: xiejiong.xj@alibaba-inc.com.*

Note that, following prior works [3, 4], we use the edit distance to measure string similarity in this paper.

As we know, traditional databases and spatial-textual analytics systems are built on disk, which makes them difficult to achieve low latency [5]. Besides, they are not able to provide high throughput when scaling to large spatial-textual data set, since they have been implemented in centralized environments [5]. Confronted with the surging data and the efficient retrieval requirement, it is natural for us to explore distributed in-memory techniques to achieve these features (i.e., low latency, high throughput and scalability). Furthermore, people tend to write SQL statements to do data analytics [5, 6], and thus it is meaningful to support spatial-keyword queries through SQL interfaces.

Spark SQL [6] is such an engine, which extends Spark (a fast distributed in-memory computing engine) to enable us to query structured data inside Spark programs. It also provides uniform data access, which allows users to connect to any data source in the same way. Recently, Xie et al. [5] presented the Simba system which extends Spark SQL to provide in-memory spatial analytics. However, none of existing distributed in-memory engines, e.g., Spark, Spark SQL and Simba, can provide native support for spatial-keyword queries. This way, for general purpose computing engines like Spark SQL, users have to rely on user defined functions to process such queries, which lacks in the underlying index optimization. Compared to the above method (i.e., using user defined functions), a better solution could be first to filter spatial results using Simba [1, 7], and then verify the textual predicates on the results of spatial filtering. However, using this solution ignores the possible pruning power of string predicates, and thus suffers from unnecessary node visits and computing cost.

Inspired by these observations, we design and implement the Skia (Spatial-keyword in-memory analytics) system. Skia employs a two-level index framework to efficiently process common spatial-keyword queries. Specifically, we propose novel global indexes (i.e., the BFR-Tree and GR-Tree, which are short for Bloom filter-R-Tree and Grams-R-Tree) for big spatial-textual data. The proposed two global indexes are both probabilistic structures for

performing first-step filtering. They can greatly reduce the number of candidate partitions while consuming small space budget. Besides, we present four novel local indexes to further reduce the query latency. The proposed BFIR-Tree combines Bloom filters [8] with spatial index structure (e.g., R-Tree) to support spatial-keyword queries. It aims to avoid performing exact keyword matching in every level of the R-Tree. We further present the dynamic CBFIR-Tree, which enables incremental maintenance while inheriting the powerful merit of BFIR-Tree. Moreover, previous work did not leverage the pruning ability of infrequent keywords. We found that infrequent keywords can greatly reduce the search space for queries with multiple keywords, which is very common in real-life scenarios. On top of this, we further propose the S2I-V index structure. As is concluded by [1], textual data analysis actually dominates spatial in most scenarios. Specifically, textual-first indexes have showed better performance in past work [1, 2]. This insight motivates us to propose the B^{ed}-first-Tree, which is a textual-first index structure. Compared with previous textual-first structures, the B^{ed}-first-Tree is based on tightly combined scheme, i.e, wielding the pruning ability of spatial filters and textual filters together. Experimental results based on large-scale datasets show that the B^{ed}-first-Tree achieves better performance than state-of-the-art solutions. Notice that the B^{ed}-first-Tree can be easily implemented based on existing B^{+}-trees [9], which are supported by almost all modern database systems [10]. Last but not least, we extend Spark SQL to provide convenient SQL programming interfaces for data analytics. Specifically, we first extend the Spark SQL to provide new keywords (including keywords for new index types, new predicates) for the spatial-keyword queries. We also add new parsing rules to the Spark SQL parser so that the SQL statements for spatial-keyword queries can be recognized and we finally implement the execution logic to finish the execution of these queries.

In a nutshell, our main contributions are as follows:

- We introduce four index structures, i.e., the BFIR-Tree, CBFIR-Tree, S2I-V and B^{ed}-first-Tree to answer four types of spatial-keyword queries, which serve as the local indexes in our framework mentioned below (Section 4).
- We propose the two-level index framework for spatial-keyword queries in distributed platforms, in which the global index structure and the four local index structures are seamlessly integrated (Section 5).
- Based on Spark, we provide the distributed solution, which are highly scalable, and achieve high throughput with low latency at the same time (Section 5).
- We extend Spark SQL to provide convenient SQL programming interfaces to perform the complex tasks with just a few lines of SQL statements (Section 6).
- We conduct experiments on large-scale datasets to demonstrate the superior performance of our solution (Section 7).

In the next section, we review previous related works, followed by introducing some preliminaries (Section 3).

2 RELATED WORK

Exact spatial-keyword query: Processing exact spatial-keyword queries has been extensively studied in recent years. Zhou et al. [11] extended the study of keyword search to spatial databases. They proposed two hybrid geo-textual indexes, i.e., the Inverted file-R^*-Tree structure and R^*-Tree-Inverted file structure. Felipe et al. [12] proposed the IR^2-Tree for efficient exact spatial-keyword queries. It integrates signature file into each node of the R-Tree [13]. Cong et al. [14] proposed the IR-Tree that combines spatial and inverted indexes tightly. They also use an R-Tree and augment each node of the R-Tree with an inverted file, which summarizes the text content of the corresponding subtree. In addition, Rocha-Junior et al. [15] suggested an S2I index structure which treats items differently according to their frequency. In S2I, frequent objects are organized into an aggregated R-Tree whereas infrequent ones are indexed by inverted lists. Different from the works mentioned above, our work integrates the more efficient Bloom filters into R-Tree, which has not been explored yet. Furthermore, we address the incremental maintenance by leveraging the dynamic Bloom filter. Last but not least, we also explore how to group multiple query keywords during searching.

Approximate keyword search: Given a string set and a query string, approximate keyword search tries to discover strings similar to the query string with respect to a given distance metric. Dozens of string similarity metrics have been proposed. Edit distance is one of the representative metrics and has been studied for decades [3, 4]. In this paper, we also use the edit distance to measure string similarity. There are two types of typical methods to solve the approximate keyword search problem: (1) the inverted list based method [16–19], and (2) the B^{ed}-Tree based method [10]. The first method constructs the inverted index for n-grams of strings, thereby one can retrieve similar candidates for the query string and verify them later. The second method is an all-purpose edit distance based index scheme that inherits the standard B^{+}-Tree structure [9]. The main idea behind B^{ed}-Tree is to index strings with the B^{+}-Tree, by mapping the string domain to integer domain. Notice that, although these works are related to ours, they are obviously different from ours, since we are interested in (approximate) spatial-keyword search, instead of approximate keyword search.

Approximate spatial-keyword query: Yao et al. [7] explored approximate spatial-keyword queries by proposing the MHR-Tree. The MHR-Tree leverages the min-wise signatures to measure the set resemblance of n-grams instead of storing huge amounts of n-grams explicitly, and thus reduces a lot of space cost. Nevertheless, it may lead to false negatives, since it is a probabilistic data structure. Alsubaiee et al. [20] suggested the LBAK-Tree to retrieve all right answers, which has a 100% recall rate. The LBAK-Tree augments a specific level of R-Tree nodes with approximate string indexes, in order to get the similar candidates of the query string. In particular, if the corresponding sub-node contains none of the candidates (during query processing), then it is excluded. Different from the above two structures which are extended from spatial-based indexes, this paper proposes the B^{ed}-first-Tree, which is extended from the textual-based index (i.e., the B^{ed}-Tree) to leverages the merit of B^{ed}-Tree.

Distributed analytics engines: Distributed in-memory computing nowadays is becoming more and more popular. Apache Spark [21] is such a fast and general engine for large-scale data processing, based on in-memory computing. Compared to Hadoop which is disk-oriented and batch processing, Spark can offer low query latency and high throughput owing to distributed memory storage and computing. These features are crucial to achieve an interactive query style. Recently, Xie et al. [5] extended Spark and proposed the distributed in-memory spatial analytics engine named Simba. Simba implements a set of spatial operations (*circle*

range, *kNN*, *distance join* and so on) for analyzing large spatial data. They have experimentally proved the superior performance of Simba, compared against other engines like SpatialHadoop [22], SpatialSpark [23]. [24] explored distributed query processing on multi-dimensional data with keywords, which focused on top-k queries and thus is different from our work.

3 Preliminaries

In this section, we first present the problem formulation, and then introduce the Bloom filters, which will be employed in our solution. For ease of reference, the frequently used notations are summarized in Table 1.

TABLE 1: Frequently used notations

Notation	Description
$\mathcal{D}$	geo-textual dataset
o_i	i-th geo-textual object of $\mathcal{D}$
ρ_i	Euclidean coordinate of o_i
γ_i	the set of keywords of o_i
Q	spatial-keyword query
Q_s	spatial predicate of Q
Q_t	textual predicate of Q
(τ, r)	query center and corresponding query radius
k	number of objects to return in $eBKQ$ or $aBKQ$ query
(s_i, d_i)	i-th query word and corresponding edit distance threshold
$dist(\rho, \tau)$	Euclidean distance between points ρ and τ
$edit(s_i, s_j)$	edit distance between string s_i and s_j
θ	retrieved geo-textual objects of each query

3.1 Problem Formulation

Formally, a geo-textual dataset $\mathcal{D}$ contains $|\mathcal{D}|$ geo-textual objects. Each object o_i consists of a point ρ_i in the Euclidean space and a set of strings γ_i. Hence, a dataset $\mathcal{D}$ with n points can be described as: $\{(\rho_1, \gamma_1), \ldots, (\rho_n, \gamma_n)\}$.

A spatial-keyword query Q consists of a spatial filter Q_s and a textual filter Q_t. As for Q_s, we focus on the classical circle range query and k nearest neighbors (kNN) query. Generally, a circle range query is defined by a pair (τ, r), in which τ is the query point and r is the radius. A kNN query can be defined by a pair (τ, k), where k denotes the number of nearest neighbors to return. A textual predicate Q_t can have two cases. In the case of exact spatial-keyword queries, Q_t can be simply defined as a set of strings. Correspondingly, we can use $\{(s_1, d_1), \ldots, (s_n, d_n)\}$ to describe Q_t when it comes to approximate spatial-keyword queries, where s_i is a query term and d_i is the associated edit distance threshold; here $i \in [1, n]$ and $d_i \in [0, |s_i|]$.

We mainly explore four kinds of spatial-keyword queries according to types of spatial predicates and textual predicates. They are formally defined below.

DEFINITION 1. *Exact Boolean Range Query ($eBRQ$): Let $dist(\rho, \tau)$ be the Euclidean distance between points ρ and τ, an $eBRQ$ $Q = \{Q_s = (\tau, r), Q_t\}$ retrieves all points (denoted as θ) in D such that,*

$$\theta = \{o_i \in \mathcal{D} \mid dist(\rho_i, \tau) \leq r \wedge Q_t \subseteq \gamma_i\}.$$

In simple terms, the $eBRQ$ retrieves points whose text content contains all keywords in Q_t within the query region.

DEFINITION 2. *Exact Boolean kNN Query ($eBKQ$): An $eBKQ$ $Q = \{Q_s = (\tau, k), Q_t\}$ retrieves a subset of points $\theta = \{o_i \in \mathcal{D} \mid \forall o_j \in \mathcal{D} - \theta, Q_t \not\subseteq \gamma_j \vee dist(\rho_j, \tau) \geq dist(\rho_i, \tau)\}$ and, $|\theta| = k$.*

An $eBKQ$ tries to retrieve k nearest neighbors of the given point τ whose text content covers Q_t.

DEFINITION 3. *Approximate Boolean Range Query (a-BRQ): Given an $aBRQ$ $Q = \{Q_s = (\tau, r), Q_t = \{(s_1, d_1), \ldots, (s_n, d_n)\}\}$, the result of Q is formulated as: $\theta = \{o_i \in \mathcal{D} \mid dist(\rho_i, \tau) \leq r \wedge (\forall s_l, d_l \in Q_t, \exists s \in \gamma_i, edit(s, s_l) \leq d_l)\}$.*

In other words, an $aBRQ$ tries to retrieve points that have similar matching strings for every query string.

DEFINITION 4. *Approximate Boolean kNN Query (a-BKQ): An $aBKQ$ $Q = \{Q_s = (\tau, k), Q_t = \{(s_1, d_1), \ldots, (s_n, d_n)\}\}$ can be concluded as: $\theta = \{o_i \in \mathcal{D} \mid \forall o_j \in \mathcal{D} - \theta, (\exists s_l, d_l \in Q_t, \forall s \in \gamma_i, edit(s, s_l) > d_l) \vee dist(\rho_j, \tau) \geq dist(\rho_i, \tau)\}$ and, $|\theta| = k$.*

The $aBKQ$ will retrieve k nearest neighbors from the query point τ whose text description contains similar words to each of the query words $\{s_1, s_2, \ldots, s_n\}$.

3.2 The Bloom filters

A Bloom filter represents a set $\mathcal{S}$ of m elements from a universe $\mathcal{U}$ by using an array of n bits (denoted as $\{\mathcal{B}[1], \ldots, \mathcal{B}[n]\}$), which are initially set to 0 [8]. The Bloom filter utilizes a group $\mathcal{H}$ of κ independent hash functions $\{h_1, \ldots, h_\kappa\}$, which independently maps each element in $\mathcal{U}$ to a random number $v \in [1, n]$. It sets the bits $\mathcal{B}[h_i(x)]$ to 1 for each element $x \in \mathcal{S}$, where $i \in [1, \kappa]$. To determine if an element $y \in \mathcal{U}$ exists in $\mathcal{S}$, it examines whether all the corresponding bits $\mathcal{B}[h_i(y)]$ ($i \in [1, \kappa]$) are set to 1. If so, there is a high possibility that y exists in $\mathcal{S}$; otherwise, y definitely does not exist in $\mathcal{S}$.

4 Spatial-Keyword Search Algorithms

In this section, we first introduce the static BFIR-Tree and dynamic CBFIR-Tree, to support exact spatial-keyword queries in a memory-based computing environment. Then, the textual-first S2I-V structure is introduced to further improve the performance. Finally, we present another textual-first structure for processing approximate spatial-keyword queries.

4.1 The BFIR-Tree

IR-Tree is a general and efficient structure for exact spatial-keyword queries [14]. Intuitively, it maintains an inverted file for every node in the R-Tree, and maps keywords to the corresponding child nodes. This way, one can leverage the inverted files to filter irrelevant child nodes and thus reduce time cost. Nonetheless, two important observations motivate us to develop more competitive structures: **Observation 1** — The IR-Tree uses inverted files in every level of R-Tree and thus every keyword will be stored multiple times, which incurs huge space cost; **Observation 2** — The IR-Tree performs exact textual matches in every level of R-Tree, which is unnecessary actually. Inspired the above observations, we present a new index structure called BFIR-Tree (Bloom Filter-IR-Tree). Our structure inherits the merit of the IR-Tree, while integrating the Bloom filters. It achieves high efficiency in both time and space.

Structure: The BFIR-Tree is somewhat similar to R-tree, yet it has two types of nodes: (1) **B-Node** is a non-leaf R-Tree node and maintains a Bloom filter index, which abstracts the corresponding textual information in the sub-nodes. If this node passes the spatial predicate, we use the Bloom filter to approximately check whether this node contains the query keywords. Notice that, if one of the query keywords is not found, the node and its corresponding subtrees will not be searched further. (2) **I-Node** is a leaf R-Tree node which stores an inverted list mapping each keyword to the spatial-keyword objects. Based on inverted lists, we can validate whether the object satisfies the textual predicates, and then get final results by computing the distance to the query point (for each passed record in the I-Node).

Note that, the B-Nodes may introduce false positives, as the Bloom filter is a probabilistic data structure. However, because of its low error rate when configured well, its negative effect can be almost neglected. In addition, multiple keywords can further reduce the false positive probability of the BFIR-Tree. To explain, assume the Bloom filter has indexed n elements, using k hash functions and m bits. Let s denote the number of query keywords, then the false positive probability can be computed as

$$Pr_{false-positive} = (1 - (1 - \frac{1}{m})^{nk})^{ks} \approx (1 - e^{\frac{-kn}{m}})^{ks}.$$

As an example, when $\frac{m}{n} = 10$ and $k = 7$, the false positive probability is just only $(0.008)^s$. Remark that, the I-Nodes can guarantee the correctness of the BFIR-Tree, which means that it has a 100% recall rate.

Construction of the BFIR-Tree: Similar to the construction of R-Tree, we build the BFIR-Tree by applying a bottom-up fashion until the root node is established. For a leaf node u, let the set of points contained in u be u_p. We can build the invert file for u by mapping each term t to a list of objects containing t. Meanwhile, we collect the vocabulary of u to construct the parent node's Bloom filter. In contrast, for a non-leaf node p, let its child entries be $\{c_1, \ldots, c_f\}$, where f is the fan-out of the R-Tree and $i \in [1, f]$. We merge the vocabulary of each entry c_i and get the vocabulary of node p. Then, we insert each item into an initialized Bloom filter (we will further introduce the setup of the Bloom filter in Section 7.1). Notice that here we do not compute the union of the children's Bloom filters, due to following considerations: (1) using the union style incurs more collisions for a unique bit in the Bloom filter, and thus leads to a high error rate in high level nodes; and (2) it needs to configure a fixed length for the Bloom filter in each level, despite of the total size of terms.

Query algorithms via the BFIR-Tree: Consider an $eBRQ$ $Q = \{Q_s = (\tau, \epsilon), Q_t\}$, here Q_t is a set of keywords. At a B-Node, the algorithm first checks whether this node satisfies Q_s, namely the spatial condition. If it is in the query region, then for each word in Q_t, we check whether it is in the node's Bloom filter. A node is pruned if one of the query terms does not exist. Otherwise, we propagate the query Q downward recursively. In the propagation, if it finally reaches a I-Node, we map each term to its corresponding objects list and then compute the intersection result of these lists to get survivals.

To answer $eBKQ$, it can be achieved by revising R-tree based kNN algorithm. Simply speaking, we also maintain a priority queue, which orders objects by their distances to the query location. Yet, we only add the objects passing textual predicates into the queue now. Also, we dequeue the objects until we finally get k results, or the queue becomes empty.

Note that, the BFIR-Tree avoids performing exact textual matches each time when we visit the nodes of R-Tree. Essentially, it prunes dissimilar child nodes first and verify the survived candidates using exact matching, and thus is fast enough. In addition, it avoids storing redundant textual information, and thus reduces a lot of space cost.

4.2 The CBFIR-Tree

The BFIR-Tree developed above is space efficient and fast enough, yet it is a static structure, as the Bloom filter cannot handle deletions at all. The main reason is that, we cannot set the bits from 1 to 0 straightforwardly (as these bits may also be hashed to by other elements). To address this dilemma, we propose using the counting Bloom filter (CBF) [8]. Compared with the traditional Bloom filter, a major difference is that, the CBF uses a small counter, instead of a single bit for each entry, which enables dynamic maintenance. With the above concepts in mind, we develop the CBFIR-Tree (known as CBF-based IR-Tree), which can be viewed as a dynamic version of the BFIR Tree. The CBFIR Tree is similar to the BFIR Tree, yet it augments a count filter, which allows us to efficiently handle updates while reserving the advantages of BFIR-Tree. Besides, to support the maintenance algorithm of the CBFIR-Tree efficiently, we use the **same size** of counting Bloom filters and the same hash functions in each level of the CBFIR-Tree.

Compared to BFIR-Tree, the CBFIR-Tree only brings small extra space cost. To explain, assume there are n elements inserted, k hash functions and m counters used, the probability that any counter is bigger or equal to j is:

$$Pr(\max(c) \geq j) \leq m(\frac{\mathrm{e} \ln 2}{j})^j.$$

As for $j = 16$ (i.e., a 4-bits counter), we have

$$Pr(max(c) \geq 16) \leq 1.37 \times 10^{-15} \times m.$$

In this case, 4 bits are really enough as the probability of overflow is tiny. Due to the resemblance between the CBFIR-Tree and BFIR-Tree, we only cover the maintenance algorithm for saving space.

Incremental maintenance algorithms: Given the object $o = (\rho, \gamma)$, we first find the leaf node l to be inserted by following the **ChooseLeaf** algorithm of R-Tree [13]. There are two cases then:

- If l has room to insert the new object, i.e., no node-splits happen, then we directly insert o into l and propagate the keywords set γ bottom-up. In the propagation, for a leaf node, we rebuild its inverted file by adding mappings from each keyword in γ to the new entry. For non-leaf nodes, we simply insert each item in γ into the corresponding count Bloom filter.
- If node-split happens in the leaf node, We will recalculate the inverted list for the two new nodes. Besides, there will be a new entry added in the parent node $\mathcal{P}$ of these two nodes. If the parent node $\mathcal{P}$ does not need to be split, we simply insert each item in γ into the corresponding count Bloom filter and propagate γ bottom-up. Otherwise, we will split the node $\mathcal{P}$ into $\mathcal{P}_1$ and $\mathcal{P}_2$. The CBF of $\mathcal{P}_1$ and $\mathcal{P}_2$ can be easily reconstructed from the inverted lists of their child nodes. If the parent node of $\mathcal{P}$ (or nodes in the higher level) needs to be split, we can quickly reconstruct the counting Bloom filters by combining the their child

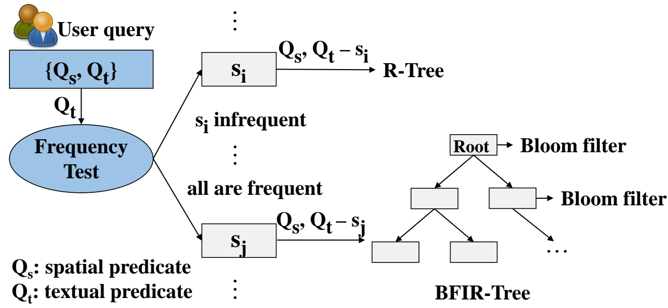


Fig. 1: The workflow of S2I-V when one frequent text item is found; or otherwise, all terms are frequent

nodes' CBFs. Notice that, we have used the same size of CBFs, so the parent node's CBF can be constructed by adding each counter of the child nodes' CBFs.

4.3 The S2I-V

It is easy to find that the above two structures are both spatial-first ones. Besides their superiorities, they also bear some limitations (explained later). The following facts motivate us to investigate the *textual-first* structure: (1) textual data analysis actually dominates spatial in most scenarios [1]; and (2) spatial-first structures leverage string matching to prune irrelevant nodes, which may be inefficient when encountering frequent text items (hereafter, frequent items for short), since we have to access many nodes (notice that a frequent item is usually owned by many nodes). Specifically, we propose a new textual first structure here, named S2I-V (Spatial Inverted Index Variant).

Our S2I-V structure inherits the merits of S2I [15], e.g., differentiating terms with different frequency. Meanwhile, it develops two important ideas: (1) it combines the pruning power of multiple keywords, especially the infrequent keyword, that enables us to search in a small candidate set; and (2) it maps frequent items to a BFIR-Tree which is very memory efficient, thereby it costs a lot less than the S2I structure.

Concretely, the S2I-V maps frequent items to a BFIR-Tree, and infrequent items to an R-Tree (notice that the frequency threshold is set manually). The S2I-V maintains a Bloom filter to determine whether a keyword is frequent. Figure 1 vividly shows the workflow of S2I-V. For instance, consider an $eBRQ$ query $Q = \{Q_s = (\tau, \epsilon), Q_t\}$, we first try to find an infrequent item in Q_t using the Bloom filter, and then use the corresponding R-Tree to process the new query $Q_{new} = \{Q_s = (\tau, \epsilon), Q_{remaining}\}$. As the infrequent item is owned by only a few objects, the new query can be processed quickly. Notice that the workflow for $eBKQ$ is similar. As for the space cost of S2I-V structure, although each infrequent keyword is mapped to its own R-Tree, it is owned by limited number of spatial-textual objects and totally the space cost of spatial part is not very huge. Next, we address how to process single and multiple keywords queries respectively.

Single keyword query: For a single keyword μ, we directly map μ to the corresponding R-Tree (if μ is infrequent) or BFIR-Tree. Notice that, we here only use the spatial part. Then, we use the circle range algorithm and kNN algorithm (performed on the R-Tree) to get results.

Multiple keywords query: For multiple keywords $Q_t = \{\mu_1, \ldots, \mu_n\}$, we first find an infrequent item μ_i. If found, we can process the new query in the corresponding R-Tree of μ_i. Specifically, for an $eBRQ$ query Q, we first use the spatial condition Q_s to get candidates, and then verify the objects containing the remaining keywords. Regarding the $eBKQ$ query, we use a priority queue ordered by distances to get the k results, which contain the query keywords. Otherwise (i.e., if not found), we get the corresponding BFIR-Tree for the first term and use it to process the new query.

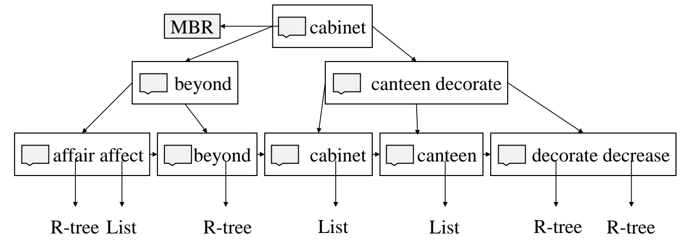


Fig. 2: The B^{ed}-first-Tree (dictionary order)

4.4 The B^{ed}-first-Tree

The state-of-the-art solutions to approximate spatial-keyword queries are all spatial-based structures. We observe that, textual data clearly dominates spatial data in most cases [1, 2], where textual-based indexes show better performance. Motivated by this, we develop a new index structure, known as the B^{ed}-first-Tree. To understand our structure, it is necessary to give a brief introduction on the B^{ed}-Tree, since our structure inherits some features of B^{ed}-Tree.

The B^{ed}-Tree is a general index structure for string processing on edit distance metric, which is based on B$^+$-Tree [10]. Compared with previous techniques like coupling n-grams with inverted lists, B^{ed}-Tree can support incremental updates efficiently. Besides, previous methods store strings redundantly, and thus leads to high space cost. Specifically, previous solution is based on the n-grams technique, which will transform one keyword into several n-grams and then map each n-gram to the keyword and the record containing this keyword. So apparently, the replication of this solution is very big. In contrast, the B^{ed}-Tree is more memory efficient, since it stores each keyword only once.

To index strings with the B$^+$-Tree, the B^{ed}-Tree applies specific transformations, mapping the string domain to integer space. The mapping function ϕ first decides the string order efficiently, which enables us to organize strings with the B$^+$-Tree efficiently, e.g., handling update operations such as "insert", "delete" and etc. Moreover, the following property enables the B$^+$-Tree to support approximate keyword search.

PROPERTY 1. *A string order ϕ is lower bounding if it efficiently returns the minimal edit distance between string q and any $s_l \in [s_i, s_j]$.*

B^{ed}-Tree stores the minimal and maximal strings s_{min} and s_{max}, which represent the boundaries of any string in the B^{ed}-Tree with respect to the string order ϕ. Assuming that the strings in the root node η are denoted as $\{s_1, \ldots, s_k\}$, it can prune each string interval in $\{(s_{min}, s_1), (s_1, s_2), \ldots, (s_{k-1}, s_k), (s_k, s_{max})\}$ by computing its minimal edit distance to the query keyword. For a survived string interval, it visits the corresponding child node and recursively traverse down the tree until reaching leaf nodes. In this case, it explicitly computes the edit distances between the

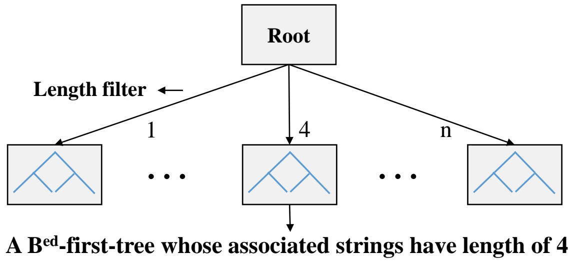


Fig. 3: The Lengthfilter-B^{ed}-first-Tree

Algorithm 1: aBRQuery

Input : A query $Q = \{Q_s, Q_t\}$ where $Q_t = \{(s_1, h_1), \ldots, (s_n, h_n)\}$; A B^{ed}-first-Tree root node r;
Output: A set of satisfied records;
1 **foreach** (s_i, h_i) in Q_t **do**
2 $\quad R_i$ = RangeQuery(Q_s, s_i, h_i, r);
3 **return** intersection result among $\{R_1, R_2, \ldots, R_n\}$;

query keyword and strings in the leaf node, in order to verify the candidates.

The main idea behind the B^{ed}-first-Tree is to incorporate the pruning power of spatial filters into the B^{ed}-Tree (notice that the B^{ed}-Tree is based on B^{+}-Tree, in which data items are all stored in the leaf nodes). Specifically, we map the frequent term to an R-Tree to organize spatial-textual objects containing this keyword, and infrequent one to a posting list (i.e., the list of records containing this keyword). During construction, for the objects in leaf nodes, we compute the minimum bounding rectangle (MBR) to represent their space region. Regarding a non-leaf node η, we group the MBRs from its children into a larger MBR. As Figure 2 shows, the B^{ed}-first-Tree is a hybrid index structure which tightly combines spatial proximity and string matching. This way, the B^{ed}-first-Tree can avoid many unnecessary string matchings which further saves the cost, compared against existing textual-based structures.

Space cost of the B^{ed}-first-Tree: Suppose the fanout of the B^{ed}-first-Tree is b, and we use a bottom-up method to bulk load the B^{ed}-first-Tree (since it is based on B^{+}-Tree, and this construction style can get all the nodes almost completely filled). Thus, if the size of the keywords set is n, the height of the tree should be $\lceil \log_b n \rceil$, and there are $b^0 + b^1 + \ldots + b^{\lceil \log_b n \rceil - 2} = \frac{b^{\lceil \log_b n \rceil} - b}{b^2 - b} = O(\frac{n-b}{b^2-b})$ non-leaf nodes. Let m be the space cost of each MBR, and γ be the average keyword length of this data set. Then, the space cost of non-leaf nodes is $O(\frac{n-b}{b^2-b} * m + \frac{n-b}{b-1} * \gamma)$. The leaf nodes need $O(n\gamma + n\sigma + nm/b)$ space to store the keywords and augmented information (i.e., the posting list or R-Tree), where σ the maximum size of the records that contain one specific keyword. Totally, the space cost is $O(\frac{nb-b}{b-1} * \gamma + n\sigma + \frac{n-1}{b-1} * m)$.

aBRQ algorithm: Our algorithm first visits the B^{ed}-first-Tree to retrieve candidates for each query keyword, as Algorithm 1 shows. The specific steps for retrieval are shown in Algorithm 2. During retrieval, as for a non-leaf node, we first compute the distance from the query point to its MBR and check whether it intersects with the circle region (Line 6). We then leverage the pruning power of string matching to choose possible child nodes (Line 7-13). In detail, if $LB(s, [s_{i-1}, s_i]) \leq \theta$, then there exist candidates in the corresponding child node of this string interval.

Algorithm 2: RangeQuery

Input : A spatial predicate Q_s; A query keyword s and its edit distance threshold θ; A B^{ed}-first-Tree root node r;
Output: A set of satisfied records;
1 Let S be a stack initialized to $\emptyset$, R be the result set initialized to $\emptyset$;
2 S.push(r, [s_{min},s_{max}]);
3 **while** $S \neq \emptyset$ **do**
4 $\quad$ Let node n and local string boundaries [s_{min},s_{max}] = S.pop();
5 $\quad$ **if** n is not a leaf node **then**
6 $\quad\quad$ **if** n's MBR overlaps with query region **then**
7 $\quad\quad\quad$ **if** $LB(s, [s_{min}, s_1] \leq \theta$ **then**
8 $\quad\quad\quad\quad$ S.push(n_1, [s_{min}, s_1]);
9 $\quad\quad\quad$ **for** i from 2 to m **do**
10 $\quad\quad\quad\quad$ **if** $LB(s, [s_{i-1}, s_i] \leq \theta$ **then**
11 $\quad\quad\quad\quad\quad$ S.push(n_i, [s_{i-1}, s_i]);
12 $\quad\quad\quad$ **if** $LB(s, [s_m, s_{max}] \leq \theta$ **then**
13 $\quad\quad\quad\quad$ S.push(n_{m+1}, [s_m, s_{max}]);
14 $\quad$ **else if** n's MBR overlaps with query region **then**
15 $\quad\quad$ **foreach** string s_i of n and $edit(s_i, s) \leq \theta$ **do**
16 $\quad\quad\quad$ Add records in the region for s_i to R;
17 **return** R;

We recursively visit the matching sub-nodes. During propagation, if we reach the leaf node, we visit the corresponding posting lists or R-Trees of matched strings to run circle range test, and then get the intermediate results (Line 15-18). The final results can be obtained by computing the intersection of intermediate results for each query keyword.

Further optimization to B^{ed}-first-Tree: An important observation is that, string length provides simple but useful information, which may help us to quickly prune strings that are not within the given edit distance. Formally, we have the following property called length filter.

PROPERTY 2. *If two strings s_1 and s_2 are within edit distance k, their lengths cannot differ by more than k.*

Based on the above, we can organize the B^{ed}-first-Tree with respect to string lengths, as Figure 3 shows. Here we use an example to show how it processes a query. Given an $aBRQ$ $Q = \{Q_s = (\tau, \epsilon), Q_t = \{(\text{"}apple\text{"}, 1)\}$, only strings whose lengths vary from 4 to 6 can pass the string predicate. Then, this query is further processed by three corresponding B^{ed}-first-Trees. Finally, we can unite results of each child tree to get the final results.

aBKQ algorithm: For an $aBKQ$ with a single keyword, we can first find the k nearest points set for each of its similar words (say as $\{K_1, K_2, \ldots, K_n\}$). Then, we search the *kNN* set among the union of $\{K_1, K_2, \ldots, K_n\}$. Regarding multiple keywords, it is somewhat complicated, since the k nearest points for a specific keyword may not contain the other query keywords. To better support $aBKQ$ for multiple keywords, we leverage the LBAK-Tree [20] to deal with the case of multiple keywords. As the LBAK-Tree is a variant of R-Tree, we can use a priority queue to

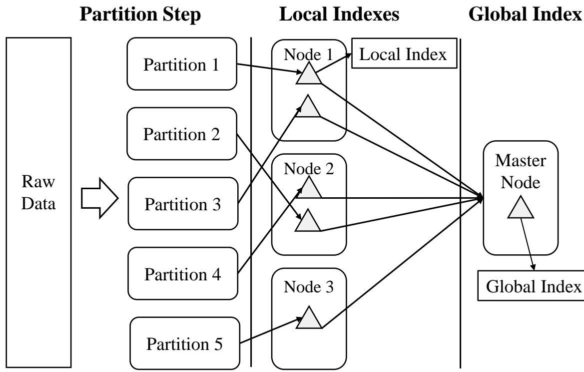


Fig. 4: The two-level index framework

order objects, which pass the textual filtering in the LBAK-Tree. This is similar to how we process $eBKQ$ via the BFIR-Tree.

5 DISTRIBUTED INDEXING

To support distributed indexing efficiently, we propose a two-level index framework that seamlessly integrates the local indexes (i.e., the indexes proposed in previous sections) and the global index (introduced later). Our framework is built on Spark, which is a widely used platform supporting distributed in-memory based environments. The global index is deployed on the master node and local indexes are deployed on slave nodes, as shown in Figure 4. For short, we refer to our distributed solution as **Skia** (Spatial-keyword in-memory analytics), which has following important merits. On one hand, local index structures like BFIR-Tree can avoid visiting irrelevant items, and thus enable us to reduce query latency. On the other hand, the global index can prune irrelevant partitions, which performs the first step of filtering and frees more CPU resources for other queries, improving query throughput significantly.

5.1 Phases of Distributed Indexing

In this subsection, we introduce our implementation of the indexing for handling spatial-keyword queries in Spark. As Figure 4 shows, the proposed algorithm includes three phases (without regard to query types and the underlying index structures), detailed as follows.

Partition: Partitioning splits data into n partitions which are mapped to each node of the cluster. Generally, a good partitioner focuses on three considerations: (1) Data Locality: nearby data items in the spatial aspect or the textual aspect should be assigned to the same partition. (2) Load Balancing: all partitions should be roughly of the same size. (3) Scalability: time cost of partitioning should be acceptable. In our paper, we propose to using a spatial-based partitioner called **STRPartitioner** to perform partitioning, due to its simplicity and proven effectiveness [5, 25]. Generally, **STRPartitioner** implements the first iteration of STR algorithm [26] to determine partition boundaries, namely minimum bounding rectangles (MBRs). Based on these MBRs, we can build a temporary R-Tree to map each object to a specific partition.

Local Indexes Construction: For each partition, we can build a specific index file. For instance, using BFIR-Tree to build local indexes to support exact spatial-keyword queries. Moreover, we can collect spatial statistics (e.g. MBRs for each partition) and textual statistics to further construct the global index. They are in the form of $(id, \textit{MBR}, \beta)$, where id identifies the partition and β represents the textual information of this partition.

Algorithm 3: BFRSearch

Input : A query $Q = \{Q_s, Q_t\}$ and $Q_t = \{s_1, \ldots, s_k\}$; A BFR-Tree root node r;
Output: A set of partition ids;

```
1  Let S be a stack initialized to ∅, R be the result set
     initialized to ∅;
2  S.push(r);
3  while S ≠ ∅ do
4      Let node n = S.pop();
5      if n is not a leaf node then
6          foreach child entry e_i of n do
7              if e_i satisfies the spatial predicate Q_s then
8                  S.push(e_i.node);
9      else
10         foreach child entry e_i of n do
11             Let flag = True;
12             if e_i satisfies the spatial predicate Q_s then
13                 foreach keyword s_i in {s_1, ..., s_k} do
14                     if s_i ∉ e_i's Bloom filter then
15                         Let flag = False and break;
16             if flag == True then R.add(i) ;
17 return R;
```

Global Index Construction: Using statistics collected in the local indexes construction, we can build a global index in the master node used to prune irrelevant partitions. Generally, it combines R-Tree, which is established by bulk loading the MBRs data, with scalable textual filtering techniques. However, using textual statistics is somewhat complicated; more details are given below.

5.2 Global Index Construction

The global index enables us to visit less partitions, which frees more CPU resources and lowers network cost in the meantime, while leading to overhead space and time cost. To get a tradeoff, we suggest that the global index should have the following features: space efficiency, fast retrieval and powerful pruning ability. Simba [5] builds a pure R-Tree as the global index to support pruning partitions in spatial analytics. Indexing 4.4 billion records, the global index only consumes 700KB. However, indexing for textual data is generally expensive both in time and space. Especially when the data size grows, the textual analysis cost will increase quickly. Thus, there should be a more elegant way to support pruning, especially for textual part. To achieve this, two techniques are presented as follows.

5.2.1 *The BFR-Tree*

The BFR-Tree is designed for global indexing when processing $eBRQ$ and $eBKQ$. The entries in its leaf nodes are in the form of $(\textit{MBR}, id, \textit{Bloom filter})$. Here we use Bloom filters to describe the text content of corresponding partition. As for non-leaf nodes, they are same to the R-Tree nodes, namely only store spatial information. In contrast to error-free hybrid indexes like BFIR-Tree or IR-Tree, the BFR-Tree is more scalable as it consumes

much less memory. In addition, it can achieve almost the same pruning performance compared to error-free techniques, since false positives rarely appear and false negatives are impossible in the BFR-Tree.

Construction of BFR-Tree: Notice that the BFR-Tree is designed for global index and thus we assume the input table R is partitioned into a set of partitions $\{R_1, \ldots, R_m\}$, where m is the number of partitions. Let the MBR and strings set of each partition be (M_i, S_i), where i corresponds to the identifier of the partition. Given all MBRs and string sets $\{(M_1, S_1), \ldots, (M_m, S_m)\}$, we construct a bloom filter B_i for S_i and get a new set $\{(M_1, B_1), \ldots, (M_m, B_m)\}$. Then we create a classic R-Tree by bulk loading this new set.

Search Algorithm of BFR-tree: Let Q be a query with a spatial predicate Q_s and a textual predicate Q_t. Algorithm 3 presents a solution for filtering partitions when processing $eBRQ$. It is also easy to revise this algorithm to support $eBKQ$ using a priority queue.

5.2.2 GR-Tree

Similar to the BFR-Tree, we can design the global index, based on the R-Tree, for approximate spatial-keyword queries. First, we introduce the widely used count filter as the property of n-grams [7].

PROPERTY 3. *If two strings s_1 and s_2 are within distance k and their n-grams are denoted as G_1 and G_2, then we have $|G_1 \cap G_2| \geq max(|s_1|, |s_2|) - 1 - (k-1) * n$.*

Inspired by the filtering technique above, we can collect the n-grams for each partition and use Bloom filters to store them respectively. Yet, there may exist the issue that a specific n-gram derives from multiple strings and thus all the n-grams of the query keyword are matched. To improve this, we employ the length filter to divide the big set into several small sets to reduce the probability of collisions.

5.3 Spatial-Keyword Queries in Spark

The proposed two-level index framework and further presented techniques enable us to explore novel algorithms for spatial-keyword queries in the context of Spark. We have mentioned four kinds of queries previously. Next, we cover their distributed solutions with respect to the spatial predicates.

5.3.1 Supporting $eBRQ$ and $aBRQ$

Although the $eBRQ$ and $aBRQ$ differs, their solutions are similar. With the aid of global index and local indexes, the solution is as follows.

Global filtering: We first use the global index to retrieve the possible partitions. For instance, we can use the BFR-Tree for $eBRQ$ to find the partitions which intersect the query area and possibly contain the query keywords. Then, we perform the local processing below for each partition respectively.

Local processing: For each partition passing the above test, we use the local index such as the BFIR-Tree to process the $eBRQ$ or the B^{ed}-first-Tree for $aBRQ$. Then, relevant results are collected together in the master node.

5.3.2 Supporting $eBKQ$ and $aBKQ$

Solutions to the $eBKQ$ and $aBKQ$ are more complicated compared to the above two. An important observation is: a partition, which fulfills the textual predicate and is closer to the query point, is more likely to contain the potential results for the $eBKQ$ or $aBKQ$. Using this observation, we can develop the algorithm as the Algorithm 4 shows.

Algorithm 4: BKQSearch

Input : $Q = \{Q_s, Q_t\}$, where $Q_s = (\tau, k)$; the global index $\mathcal{G}$; each partition's local indexes $\{\mathcal{L}_1, \ldots, \mathcal{L}_n\}$
Output: k nearest records that satisfy Q_t

```
1  Use global index G to find P = {p_1, p_2, ..., p_k}, which
   are k nearest partitions to τ that satisfy Q_t, and let R =
   ∅;
2  foreach partition p_i ∈ P do
3      R_i ← k nearest records that satisfy Q_t based on L_i;
4      R ← R ∪ R_i;
5  if |R| ≥ k then
6      μ ← the k-th smallest distance to τ in R;
7      P_new ← partitions whose minimum distances to τ are
         at most μ and they satisfy Q_t;
8      if P_new − P ≠ ∅ then
9          foreach partition p_j ∈ (P_new − P) do
10             R_j ← k nearest records that satisfy Q_t based
                 on L_j;
11             R ← R ∪ R_j;
12     Sort R by distance to τ and return k nearest records;
13 else
14     foreach partition p_j ∉ P do
15         R_j ← k nearest records that satisfy Q_t based on
             L_j;
16         R ← R ∪ R_j;
17     Sort R by distance to τ and return k nearest records;
```

Step 1: Based on the global index, we first find the k nearest partitions which pass the string test (Line 1). For the retrieved k partitions, we can use corresponding local indexes to further process the query, and collect these results to the driver program (Line 2-4). If k or more than k results are collected, we will use the tighter bound in Step 2. Otherwise, Step 3 is performed.

Step 2: If k or more than k results are collected, we will calculate the k-th minimum distance from query point to the candidates collected in step 2 as the radius μ (Line 5-6). Then, we leverage the global index again to find partitions, which pass textual predicates, and whose MBRs intersect with the circle centered at the query point with the radius μ (Line 7). If the newly retrieved partitions are the same, then we order the candidates in step 1 by distances to the query point, and take k nearest records. Otherwise, we shall retrieve results from the newly added partitions, and merge the results with the ones from Step 1 and then take k nearest records (Line 8-12).

Step 3: If less than k results are returned, the remaining partitions should further process the query, in order to assure the k nearest items are retrieved correctly. We will merge the results with the ones from Step 1 and then take k nearest records (Line 14-17).

6 IMPLEMENTATION IN SPARK

In this section, we introduce the details of our implementation based on Spark SQL, i.e., the designed SQL programming inter-

faces, memory management and optimization rules.

6.1 Programming Interfaces

Users tend to organize data in databases and then write simple SQL statements to do analytics. Inspired by this observation, we extend Spark SQL and wield interfaces introduced by Simba to provide an easy way for analyzing spatial-textual data. To put it clear, we proceed to illustrate them with examples.

Point. A spatial-keyword query needs to specify the query location and thus we need to introduce a new data type called `Point`. Using the keyword `Point`, users can express a multi-dimensional position. For example, users can describe a restaurant location by using `Point(16.23, 21.46)`.

Spatial-keyword predicates. We extend SQL to provide rich spatial-keyword predicates. As mentioned above, there are four kinds of spatial-keyword queries and here we give the programming interfaces for each of them.

eBRQ. For instance, to find places near a point of interest or POI for short (say within distance 20) whose text description contains "relish" and "coffee", users can use command:

```
SELECT * FROM table1 WHERE POINT(x, y) IN
CIRCLERANGE(POINT(10, 6), 20) AND
CONTAINS(s, "relish", "coffee").
```

eBKQ. The query below can ask for 5 nearest neighbors of a POI with keyword "cake" and "lemon" from table `table1`:

```
SELECT * FROM table1 WHERE POINT(x, y) IN
KNN(POINT(12, 10), 5) AND
CONTAINS(s,  "cake", "lemon").
```

aBRQ. An approximate Boolean range query as follows asks for places near a POI (say within distance 6) whose text description contains a keyword whose edit distance from "apple" is within 1 and another keyword having an edit distance within 2 from "banana":

```
SELECT * FROM table1 WHERE POINT(x, y) IN
CIRCLERANGE(POINT(14, 17), 6) AND
s LIKE (("apple", 1), ("banana", 2)).
```

aBKQ. E.g., the following command retrieves 5 nearest neighbors of a POI whose text description contains a keyword having an edit distance within 3 from "electron" from table `table1`:

```
SELECT * FROM table1 WHERE POINT(x, y) IN
KNN(POINT(13.2, 18.7), 5) AND
s LIKE (("electron", 3)).
```

Index management. Users can specify the index structure by using `"USE INDEXTYPE"` where `"INDEXTYPE"` is the keyword for a specific index structure. To create a BFIR-Tree index called `bfirIndex` on attributes `x`, `y` and `s` for table `street`, you can use SQL command:

```
CREATE INDEX bfirIndex ON street(x, y, s) USE BFIRTREE.
```

6.2 Memory Management

Caching indexes in memory is really important for real-time analytics. In our experiments, we found that the average query latency for queries would be several minutes, when indexes are partially cached in disk due to insufficient memory. Besides, indexing for spatial-textual objects actually costs much memory, especially for the textual part. However, current Spark-based in-memory analytics engines, like Simba [5] or LocationSpark [27], all store and manipulate indexes through underlying Java Virtual Machines (JVMs) [28]. This incurs two performance issues: (1) With large memory consumption, the JVM is easy to be trapped in frequent garbage collection and even Full Garbage Collection (generally costs several minutes) [28]. This hurts the performance a lot. (2) Storing indexes as JVM objects directly can consume more memory due to extra object headers. In Skia, we decide to manage our indexes through off-heap memory space. Indexes are serialized into bytes and stored in off-heap memory. This can entirely solve the problem of GC and further reduce the memory cost of indexing.

6.3 Query Optimization

This section covers three optimizations, and we address them respectively: (1) Intuitively, we can reduce the lengths of lists to union if we process intersection first. For instance, for a query $(A \vee B) \wedge C$, it can be rewritten as $(A \wedge C) \vee (B \wedge C)$. Moreover, [29] has further proved that if the length of C's result is shorter than A's or B's, then the latter costs less than the original query; otherwise, the cost will be at most $\frac{5}{3}$ times the original query, which is less likely to happen. Based on this, we first rewrite the query into **Disjunctive Normal Form (DNF)**. (2) It is not hard to understand that, not all predicates in the given query can be optimized by indexes. For example, given a DNF query $(A \wedge B \wedge C) \vee (D \wedge E) \vee (D \wedge F)$, it is possible that only A, B and D can be optimized by indexes. To address this dilemma, we form a new query by temporally abandoning the predicates that cannot be optimized. E.g., in the above example we form a new query $(A \wedge B) \vee D$. (3) It is possible that there are some relevant predicates, and a straightforward process could be inefficient. For example, if A and B are both Exact Boolean Range queries, and the query circle range of A is completely included by B's, then the results for query $A \wedge B$ should all be in the range of A's query circle. Besides, the results for query $A \wedge B$ should contain both the query keywords of A and B. Thereby, we could combine relevant predicates and then process the new query. Continue the above example, if A and B are relevant predicates, then we can further rewrite the new query as A' to reduce the search space. This way, one can wield the indexes to calculate the $A' \vee D$ and filter the intermediate result using the abandoned predicates C, E and F.

7 EXPERIMENT

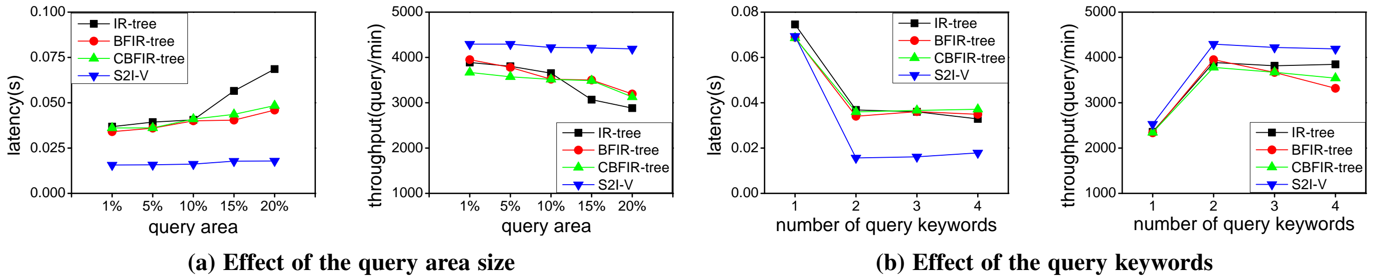
7.1 Experiment Setup

We implement our index techniques and spatial-keyword operations based on Spark SQL. Our experiments are performed on a cluster of 17 nodes with three configurations: (1) 8 machines with a 6-core Intel Xeon E5-2603 v3 1.60GHz processor and 20GB RAM; (2) 2 machines with a 6-core Intel Xeon E5-2620 2.00GHz processor and 56GB RAM; (3) 7 machines with a 6-core Intel Xeon E5-2609 1.90GHz processor and 16GB RAM. One machine of type (2) is selected as the master node and the others as slave nodes. Each slave node uses 15GB memory and all the available 6 cores for further computing. All the nodes are running on Ubuntu 14.04.2 LTS with Hadoop 2.4.1 and Spark 1.3.0.

In our experiments, two real world datasets are used: (1) **OSM** dataset contains 30 million records. The spatial part is obtained from the OpenStreetMap project [30] consisting of two-dimensional coordinates. Each point augments with strings extracted from documents collected by the SNAP [31]. The average keywords size for each record is 16.5 and the average keyword length is 9.6. (2) **TX-CA** dataset contains 26 million points, including 14 million points from the TX dataset and 12 million

TABLE 2: The space cost of index structures on OSM dataset

Index structure	IR-Tree	BFIR-Tree	CBFIR-Tree	S2I-V	LBAK-Tree	B^{ed}-first-Tree
Space cost(GB)	59.8	38.9	43.6	40.4	28.8	24.2



(a) Effect of the query area size

(b) Effect of the query keywords

Fig. 5: Comparison among index structures for eBRQ on TX-CA dataset

points from the CA dataset. These points represent the real road network and streets of Texas and California [30]. We combine each point with its state, county and town names as associated strings. The average keywords size for each record is 5.2 and the average keyword length is 7.0. Notice that although each node uses 15GB memory, Spark platform has reserved some memory for its execution. Due to limitation of available memory for caching indexes, these two datasets are appropriate for our experiments.

Following prior works [5, 22], we mainly focus on two metrics to evaluate the performance, namely the *throughput* and *latency*. Specifically, we start 10 threads continually issuing queries and totally 500 queries are tested each time, in order to calculate the average latency. The throughput is measured by the number of requests handled per minute.

Regarding the query generation, we randomly select one record from the given dataset, and use its point as the query location. To learn the impact of query keywords, we randomly choose a specified number of keywords from each record. This way, one result can be retrieved at least. Besides, we also vary the query radius to study how query area affects the performance. Specifically, the query area is a percentage over the entire area covered by the dataset points, whose default value is 1%. In our experiments, we vary the percentage among $\{1\%, 5\%, 10\%, 15\%, 20\%\}$. For approximate spatial-keyword queries, the edit distance threshold varies from 1 to 3. Notice that, a bigger threshold than 3 can introduce too many meaningless results. The default HDFS block size is 64MB, and we divide the data set into 150 partitions in default. As for the setup of Bloom filters (used for the BFR-Tree and the BFIR-Tree), MD5 hash functions are used. To minimize the probability of false positives of Bloom filters, it is important to select appropriate number of hash functions. Following prior work [8], we set number of hash functions as $\kappa = \frac{m}{n} \ln 2$ ($\frac{m}{n}$ is the number of bits used for each element). In this case, the false positive rate is $(\frac{1}{2})^{\kappa} = (0.6185)^{\frac{m}{n}}$. We have used 8 bits for each element (i.e., the $\frac{m}{n}$ is fixed to 8) to achieve predictable and acceptable false positive possibility. Further, we can quickly know the Bloom filter size m (notice that, the indexed elements size n is already known during the construction of Bloom filters).

In our experiments, we first investigate the performance of proposed index structures. Specifically, we have implemented the proposed index structures as local indexes in Skia and have done the experiments under distributed environments. Then we compare our distributed solution (i.e., Skia) against state-of-the-art analytic engines. We also explore the scalability w.r.t the data size and cluster size. Lastly, the effectiveness of the proposed global index scheme is investigated.

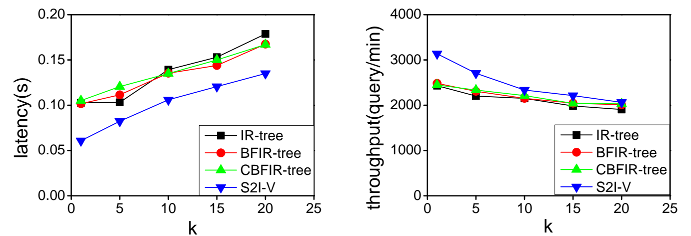


Fig. 6: Comparison among index structures for eBKQ on TX-CA dataset

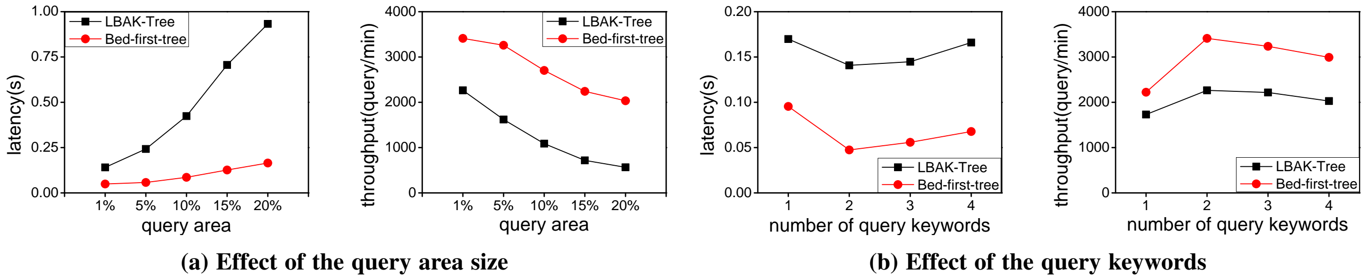
7.2 Comparison among Index Structures

In this subsection, we aim to compare the performance of our proposed index structures and existing ones in distributed environments. To begin with, we vary the query parameters for exact spatial-keyword queries and observe the performance of each index structure. Then we turn to the comparison on approximate spatial-keyword queries.

7.2.1 Performance of exact spatial-keyword queries

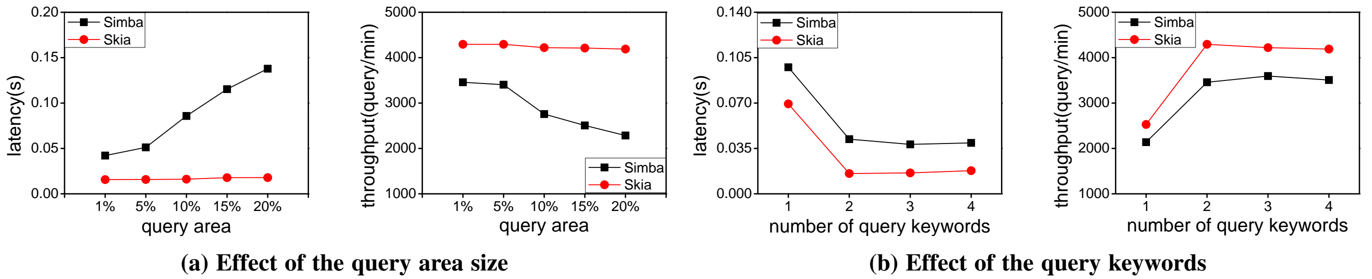
$eBRQ$ **queries.** Figure 5a shows the effect of query area by varying the query percentage. The four structures all witnessed rather slow degradation in performance, i.e., both in system throughput and average latency. The bigger query area introduces higher cost but due to the additional pruning power of string predicates, the cost just increases slightly. Besides, it is obvious that our proposed indexes achieve better scalability and performance than the IR-Tree. Especially, the S2I-V is more tolerant to the variance of query area.

Next, Figure 5b illustrates the impact of query keywords. As we can see, our proposed BFIR-Tree performs as good as the IR-Tree. Notice that the BFIR-Tree outperforms the IR-Tree a lot in space budget (refer to Table 2). Besides, the S2I-V stands out when the number of keywords increases. This is mainly because that, when the number of query keywords increase, the S2I-V performs better by leveraging the pruning ability of infrequent items.



(a) Effect of the query area size (b) Effect of the query keywords

Fig. 7: Comparison among index structures for aBRQ on OSM dataset



(a) Effect of the query area size (b) Effect of the query keywords

Fig. 8: Comparison with Simba for eBRQ on TX-CA dataset

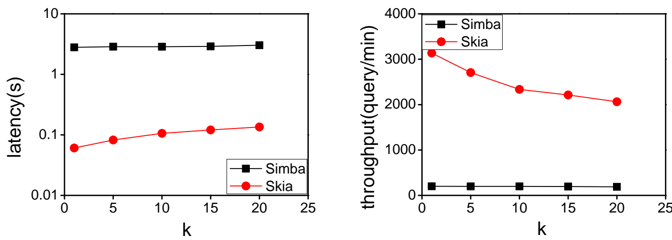


Fig. 9: Comparison with Simba for eBKQ on TX-CA dataset

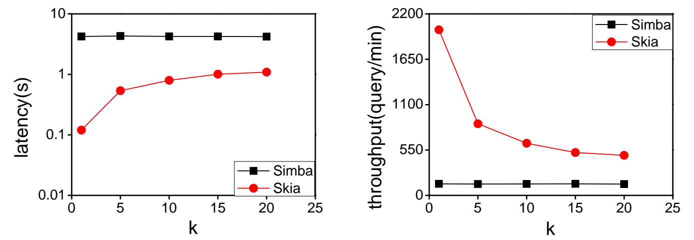


Fig. 10: Comparison with Simba for aBKQ on OSM dataset

$eBKQ$ **queries.** Figure 6 shows the performance of $eBKQ$ queries. We vary k from 1 to 20 and explore its influence. As k increases, all the four techniques witness small drop in performance. The results indicate that a first probing step (recall Section 5) is effective in most times, which avoids visiting every partition to get the final results.

7.2.2 Performance of approximate spatial-keyword queries

Figure 7a shows the impact of query area sizes. It can be seen that, the B^{ed}-first-Tree outperforms the LBAK-Tree a little when query area is small. Whereas, the B^{ed}-first-Tree is more scalable and stable when the query area enlarges.

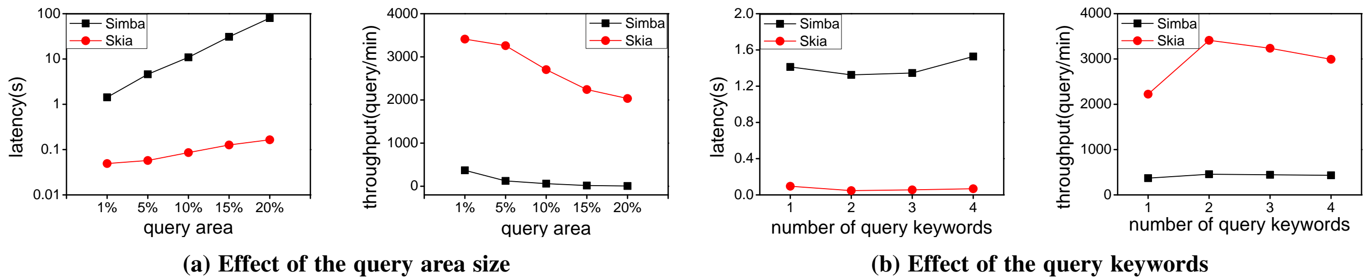
Figure 7b shows the impact of query keyword sizes. Interestingly, as the number of query keywords increases, the performance of both first grows and then decreases continually. This could be due to that, the query retrieves a lot more results with only one query keyword assigned. When two query keywords are assigned, some nodes become irrelevant and thus less visits happen, which leads to smaller cost. However, the cost of approximate string search will become the main issue when the query keywords size continues to increase. Thus we witness a decreasing trend of performance when keywords size is bigger than 2.

7.3 Comparison against Simba

To the best of our knowledge, there does not exist available distributed *spatial-keyword analytics engines* that support these four spatial-keyword queries. A straightforward idea is to extend existing *spatial analytics engines* to support the four queries. Generally, these engines build their indexes based on R-Tree (or similar structures), and it is not hard to revise them with the final phase of textual filtering (e.g., we can augment the leaf nodes of the R-Tree with textual indexes). As Simba [5] has proved its superior performance compared to other *spatial analytics engines*, we extend Simba to support the aforementioned four queries, and then we compare Skia against the extended Simba (Simba for short). We implement Skia in the same environment as Simba. Details of comparison are given below.

$eBRQ$ **queries.** Figure 8 shows the performance of exact spatial-keyword range queries. Skia outperforms Simba slightly when the query area is small. This is mainly because that, we just need to visit a small set of nodes and the pruning power of text cannot be utilized fully. However, the performance of Simba drops quickly when the query area enlarges, since many irrelevant nodes are visited in Simba. In contrast, Skia still maintains powerful performance, as Skia's textual pruning ability starts to make a difference.

$eBKQ$ **queries.** Figure 9 shows the performance of $eBKQ$



(a) Effect of the query area size

(b) Effect of the query keywords

Fig. 11: Comparison with Simba for aBRQ on OSM dataset

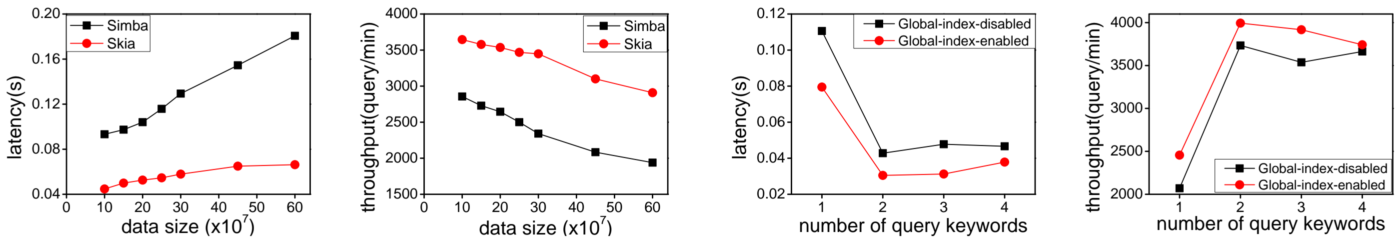


Fig. 12: Effect of the data size for eBRQ

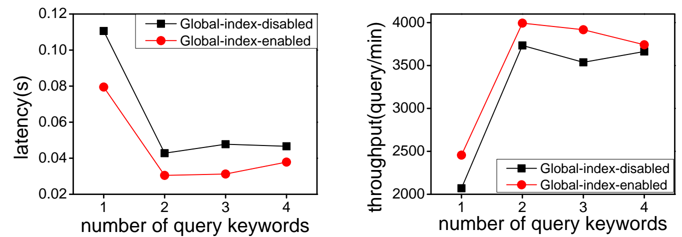
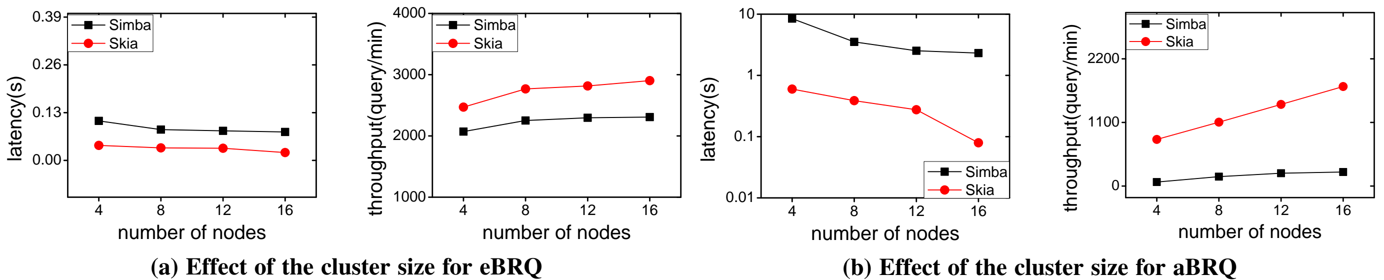


Fig. 13: Effect of the global index for eBRQ



(a) Effect of the cluster size for eBRQ

(b) Effect of the cluster size for aBRQ

Fig. 14: The impact of cluster size

queries. Skia apparently dominates Simba here. This is because that, Simba needs to visit many irrelevant partitions, without a powerful global index. Besides, it suffers from the inefficient local indexes.

$aBRQ$ **queries.** For approximate spatial-keyword queries, it can be seen from Figure 11 that, Skia outperforms Simba by about one order of magnitude. The widening gap of performance is due to the expensive cost of approximate string matching, and that our proposed B^{ed}-first-Tree remarkably accelerates the process. The performance of both Skia and Simba drops when the query area enlarges. However, Skia achieves better scalability with a slower dropping in performance.

$aBKQ$ **queries.** As Figure 10 shows, the performance of Skia drops when k enlarges. This is because that, we need to visit more partitions when k increase. However, Skia still performs better than Simba here, this is mainly due to the help of efficient local indexes. Compared to results of $eBKQ$ queries, performance of Skia deteriorates quicker, as processing approximate spatial-keyword queries is more expensive.

The impact of data size: Figure 12 shows that the performance of both solutions drops, when data size increases from 10 million to 60 million records (2×OSM). However, Skia achieves better scalability as it drops much slower. It proves that our solution can scale to larger data set.

Effectiveness of the global index: We compare two schemes: spatial index only (e.g., a single R-Tree) and spatial-textual index (e.g., the BFR-Tree). During the comparison, the same local index (S2I-V) is used. As Figure 13 depicts, the hybrid global index (i.e., spatial-textual index) apparently accelerates the searching process. This demonstrates that, although our partitioning scheme is text-insensitive, the augmented textual index can actually enhance the pruning ability.

The impact of cluster size: Experiment results for $eBRQ$ and $aBRQ$ are illustrated by Figure 14. In detail, the performance of both queries increases as the cluster size enlarges. This is intuitive and complies to our expectation, that the larger cluster size introduces stronger computing ability. Besides, in our experiments, we found that the cluster size has more impact on the performance of approximate spatial-keyword queries.

Comparison with Simba using the same global index: We also want to verify that Skia is more efficient than Simba even if the effectiveness of our proposed global index is compromised. As Simba uses the R-Tree as the global index by default, we also set Skia's global index to R-Tree. Under this setup, the comparison result is showed in Figure 15. As it depicts, the performance of Skia is still better than Simba because of the stronger pruning ability of local indexes in Skia.

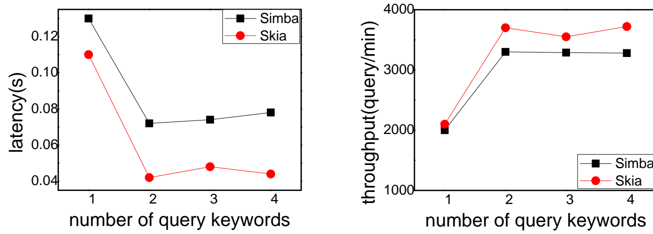


Fig. 15: Comparison with Simba (Use R-Tree as global index)

8 Conclusions

This paper proposed a series of index structures including the BFIR-Tree, CBFIR-Tree, S2I-V and the B^{ed}-first-Tree to answer spatial-keyword queries. We developed a two-level index framework to seamlessly integrate the global indexes (i.e., the BFR-Tree and GR-Tree) and local indexes together. We also designed distributed processing algorithms that fully wield the proposed framework and indexes. We conducted extensive experiments under a general distributed environment, showing that our proposed indexes and distributed solution achieve superior performance, compared against state-of-the-art techniques.

References

[1] M. Christoforaki, J. He, C. Dimopoulos, A. Markowetz, and T. Suel, "Text vs. space: efficient geo-search query processing," in *CIKM*, 2011, pp. 423–432.

[2] L. Chen, G. Cong, C. S. Jensen, and D. Wu, "Spatial keyword query processing: an experimental evaluation," in *VLDB*, 2013, pp. 217–228.

[3] A. Arasu, S. Chaudhuri, K. Ganjam, and R. Kaushik, "Incorporating string transformations in record matching," in *SIGMOD*, 2008, pp. 1231–1234.

[4] N. Koudas, A. Marathe, and D. Srivastava, "Flexible string matching against large databases in practice," in *VLDB*, 2004, pp. 1078–1086.

[5] D. Xie, F. Li, B. Yao, G. Li, L. Zhou, and M. Guo, "Simba: Efficient in-memory spatial analytics," in *SIGMOD*, 2016, pp. 1071–1085.

[6] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, and A. Ghodsi, "Spark SQL: Relational data processing in Spark," in *SIGMOD*, 2015, pp. 1383–1394.

[7] B. Yao, F. Li, M. Hadjieleftheriou, and K. Hou, "Approximate string search in spatial databases," in *ICDE*, 2010, pp. 545–556.

[8] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: a scalable wide-area web cache sharing protocol," *IEEE/ACM Transactions on Networking*, vol. 8, no. 3, pp. 281–293, 2000.

[9] D. Comer, "Ubiquitous B-tree," *ACM Computing Surveys (CSUR)*, vol. 11, no. 2, pp. 121–137, 1979.

[10] Z. Zhang, M. Hadjieleftheriou, B. C. Ooi, and D. Srivastava, "Bed-tree: an all-purpose index structure for string similarity search based on edit distance," in *SIGMOD*, 2010, pp. 915–926.

[11] Y. Zhou, X. Xie, C. Wang, Y. Gong, and W.-Y. Ma, "Hybrid index structures for location-based web search," in *CIKM*, 2005, pp. 155–162.

[12] I. De Felipe, V. Hristidis, and N. Rishe, "Keyword search on spatial databases," in *ICDE*, 2008, pp. 656–665.

[13] A. Guttman, "R-trees: a dynamic index structure for spatial searching," in *SIGMOD*, 1984, pp. 47–57.

[14] G. Cong, C. S. Jensen, and D. Wu, "Efficient retrieval of the top-K most relevant spatial web objects," in *VLDB*, 2009, pp. 337–348.

[15] J. B. Rocha-Junior, O. Gkorgkas, S. Jonassen, and K. Nørvåg, "Efficient processing of top-K spatial keyword queries," in *SSTD*, 2011, pp. 205–222.

[16] A. Behm, S. Ji, C. Li, and J. Lu, "Space-constrained gram-based indexing for efficient approximate string search," in *ICDE*, 2009, pp. 604–615.

[17] M. Hadjieleftheriou, A. Chandel, N. Koudas, and D. Srivastava, "Fast indexes and algorithms for set similarity selection queries," in *ICDE*, 2008, pp. 267–276.

[18] C. Li, J. Lu, and Y. Lu, "Efficient merging and filtering algorithms for approximate string searches," in *ICDE*, 2008, pp. 257–266.

[19] S. Sarawagi and A. Kirpal, "Efficient set joins on similarity predicates," in *SIGMOD*, 2004, pp. 743–754.

[20] S. Alsubaiee, A. Behm, and C. Li, "Supporting location-based approximate-keyword queries," in *SIGSPATIAL*, 2010, pp. 61–70.

[21] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," *HotCloud*, vol. 10, no. 10, pp. 10–10, 2010.

[22] A. Eldawy and M. F. Mokbel, "Spatialhadoop: A mapreduce framework for spatial data," in *ICDE*, 2015, pp. 1352–1363.

[23] S. You, J. Zhang, and G. Le, "Large-scale spatial join query processing in cloud," in *ICDE*, 2015, pp. 34–41.

[24] D. Amagata, T. Hara, and S. Nishio, "Distributed top-k query processing on multi-dimensional data with keywords," in *SSDBM*. ACM, 2015, p. 10.

[25] A. Eldawy, L. Alarabi, and M. F. Mokbel, "Spatial partitioning techniques in spatialhadoop," *PVLDB*, vol. 8, no. 12, pp. 1602–1605, 2015.

[26] S. T. Leutenegger, M. A. Lopez, and J. Edgington, "Str: A simple and efficient algorithm for R-tree packing," in *ICDE*, 1997, pp. 497–506.

[27] M. Tang, Q. M. Malluhi, Q. M. Malluhi, M. Ouzzani, and W. G. Aref, "Locationspark: a distributed in-memory data management system for big spatial data," *PVLDB*, vol. 9, no. 13, pp. 1565–1568, 2016.

[28] "JVM Introduction," https://docs.oracle.com/javase/specs/jvms/se7/html/index.html/.

[29] T. Lee, J. W. Park, S. Lee, S. W. Hwang, S. Elnikety, and Y. He, "Processing and optimizing main memory spatial-keyword queries," *PVLDB*, vol. 9, no. 3, pp. 132–143, 2015.

[30] "Openstreetmap project," http://www.openstreetmap.org.

[31] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.