Joint Top-K Spatial Keyword Query Processing

Dingming Wu, Man Lung Yiu, Gao Cong, and Christian S. Jensen, *Fellow, IEEE*

Abstract—Web users and content are increasingly being geo-positioned, and increased focus is being given to serving local content in response to web queries. This development calls for spatial keyword queries that take into account both the locations and textual descriptions of content. We study the efficient, joint processing of multiple top-k spatial keyword queries. Such joint processing is attractive during high query loads and also occurs when multiple queries are used to obfuscate a user's true query. We propose a novel algorithm and index structure for the joint processing of top-k spatial keyword queries. Empirical studies show that the proposed solution is efficient on real datasets. We also offer analytical studies on synthetic datasets to demonstrate the efficiency of the proposed solution.

Index Terms—H.2.4.k Spatial databases, H.2.4.n Textual databases.

1 Introduction

A range of technologies combine to afford the web and its users a geographical dimension. Geo-positioning technologies such as GPS and Wi-Fi and cellular geo-location services, e.g., as offered by Skyhook, Google, and Spotigo, are being used increasingly; and different geo-coding technologies enable the tagging of web content with positions. Studies [21] suggest that some 20% of all web queries from desktop users exhibit local intent, i.e., query for local content. The percentage is likely to be higher for mobile users.

This renders so-called *spatial keyword queries* [1]–[3], [6], [26], [27] important. Such queries take a location and a set of keywords as arguments and return the content that best matches these arguments. Spatial keyword queries are important in local search services such as those offered by Google Maps and a variety of yellow pages, where they enable search for, e.g., nearby restaurants or cafes that serve a particular type of food. Travel sites such as TripAdvisor and TravellersPoint may use spatial keyword queries to find nearby hotels with particular facilities.

As an example, in Figure 1, a service provider's database $\mathcal{D}$ stores the spatial locations and textual descriptions (sets of keywords) of restaurants p_1, p_2, p_3, and p_4. For instance, restaurant p_1 is described by the keywords: 'pizza' and 'grill.' User q_1 (shown as a shaded dot) issues the following query: Find the nearest restaurant that serves 'curry' and 'sushi.' The service returns restaurant p_2, which is the closest one that contains all the keywords in q_1.

- *D. Wu is with the Department of Computer Science, Aalborg University, DK-9220, Aalborg, Denmark. E-mail: dingming@cs.aau.dk*
- *M. L. Yiu is with the Department of Computing, Hong Kong Polytechnic University, Hung Hom, Kowloon, Hong Kong. E-mail: csmlyiu@comp.polyu.edu.hk*
- *G. Cong is with the School of Computer Engineering, Nanyang Technological University, Singapore. E-mail: gaocong@ntu.edu.sg*
- *Christian S. Jensen is with the Department of Computer Science, Aarhus University, DK-8200 Aarhus, Denmark. E-mail: csj@cs.au.dk*

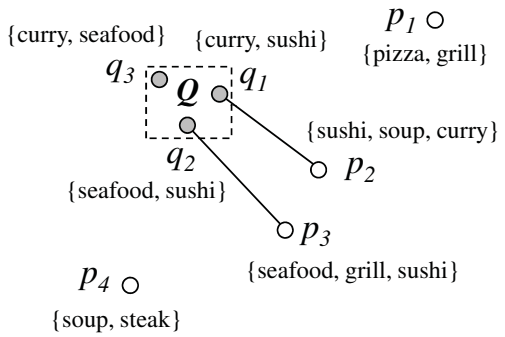


Fig. 1. Top-k Spatial Keyword Queries

A top-k spatial keyword query retrieves k objects that are closest to the query location and that contain all the argument keywords [3]. We study the joint processing of sets of such queries.

Consider again Figure 1 where the dashed rectangle represents a theater. After a concert, users q_1, q_2, and q_3 each wish to find a nearest restaurant ($k = 1$) that matches their preferences. User q_1 prefers 'curry' and 'sushi,' user q_2 prefers 'seafood' and 'sushi,' and user q_3 prefers 'curry' and 'seafood.' The result returned to q_1 is p_2, as this is the nearest object that contains all keywords of this user's query. Similarly, the result returned to q_2 is p_3, and the result returned to q_3 is empty, as no object contains 'curry' and 'seafood.'

These three queries can be processed jointly as a single joint top-k spatial keyword query Q that consists of three subqueries. The joint processing is motivated by two main applications.

Application I: Multiple Query Optimization.
Multiple query optimization is well-studied. Existing work can be roughly divided into three categories.

Grouping/partitioning a large set of queries. Papadopoulos et al. [17] use a space filling curve to group queries so as to improve the overall performance of processing all queries. Zhang et al. [28] study the processing of multiple nearest neighbor queries; they propose R-tree-based solutions and heuristics for the grouping of queries. However, the joint query we consider is actually one group of queries. We thus aim to compute

one group of queries efficiently. Any grouping/partitioning approach can be applied before our algorithm.

Caching historical information for future queries. Hu et al. [9] cache the result objects as well as the index that supports these objects as the results so as to optimize query response time. Zheng et al. [29] consider a scenario where objects are stationary while queries are mobile. They develop a semantic caching scheme that records a cached item (object) as well as its valid range (Voronoi cell). However, their approach cannot be directly applied to our problem because their valid range concept ignores the query keywords in our problem. In our experiments, traditional caching, i.e., LRU buffering, is considered as a competitor to our proposals.

Techniques for the efficient processing of one group of queries [8], [10], [18], [22]. For example, sub-expressions shared among a set of SQL queries can be evaluated once and then subsequently reused. This reduces the cost when compared to processing each query separately. None of these existing works consider keyword-based queries. However, we adopt the same general philosophy and aim to share computation across multiple queries.

With a high load of queries, joint processing is possible and can contribute to offering robustness to peak loads. For example, many users who attend or follow the same event, e.g., a sporting event, may issue similar queries that can be processed jointly with significant performance gains when compared to one-at-a-time processing. Furthermore, a variety of services are seeing very large and increasing volumes of queries, e.g., status updates and buzz posts in Facebook, tweets in Twitter, and web queries in Bing, Google, and Yahoo! Specifically, Google is reported to currently receive on average some 34,000 queries per second. This suggests a need for means of joint query processing.

The top-k spatial keyword query supports a range of important location based services. According to a new report by Juniper Research[1], revenues from mobile location-based services are expected to reach more than $12.7 billion by 2014, driven by smartphone proliferation, a surge in application storefront launches, and developments in hybrid positioning technologies. We may anticipate that the high load of top-k spatial keyword queries will become a challenge at the server side in the future.

Application II: Privacy-Aware Querying Support.
A service provider may not be fully trusted by users. For example, service providers are vulnerable to hackers, and authorities can gain access to a service provider's query logs by means of a search warrant.

A fake-query approach has been proposed that offers location privacy by hiding the true query among multiple fake queries [4], [12], [13]. However, the keywords in a query may also disclose sensitive information about a user, e.g., of a medical, financial, political, or religious nature [19]. The joint processing of top-k spatial keyword queries enables extension of the fake-query approach to also offer keyword privacy.

Specifically, the user submits a set of queries $Q = \{q_1, q_2, \cdots, q_C\}$ to the server such that q_i is the user's original query. In regards to keyword privacy, query Q is said to satisfy C*-plausible deniability* [15] if each subquery of Q has same chance of being the original query and different subqueries belong to different topics. Murugesan et al. [15] study how to generate a set Q of keyword queries such that C-plausible deniability is achieved. We aim at efficient query evaluation of a set of spatial keyword queries.

1. https://www.juniperresearch.com/reports/mobile_location_based_services

Challenges and Contributions.
While efficient for individual queries, existing top-k spatial keyword query processing techniques [2], [3], [26], [30] are inefficient for the joint processing of such queries. Thus, better techniques are needed. These should preferably be generic and applicable to a variety of tree-based index structures for spatial keyword data [2], [3], [26], [30].

The paper contributes as follows. First, we formulate the joint top-k spatial keyword query processing problem and identify applications (Sections 1 and 2). Second, we propose a generic group-based algorithm (GROUP) for the joint processing of top-k spatial keyword queries that effectively shares the processing among queries and also introduce a basic algorithm ITERATE in Section 3. Third, we present a new index structure for efficient query processing in Section 4. Fourth, we develop cost models for two representative indexes in Section 5. Fifth, we offer empirical insight into the performance of the proposed algorithm and index structure, in Section 6. We review existing indexes in Section 7 and conclude and offer research directions in Section 8.

2 PROBLEM STATEMENT

Let $\mathcal{D}$ be a dataset in which each object $p \in \mathcal{D}$ is a pair (λ, ψ) of a spatial location $p.\lambda$ and a textual description $p.\psi$ (e.g., the facilities and menu of a restaurant).

Similarly, a spatial keyword query [3] $q = \langle \lambda, \psi \rangle$ has two components, where $q.\lambda$ is a spatial location and $q.\psi$ is a set of keywords. The answer to query q is a list of k objects that are in ascending order of their distance to the query location $q.\lambda$ and whose descriptions contain the set of query keywords $q.\psi$.

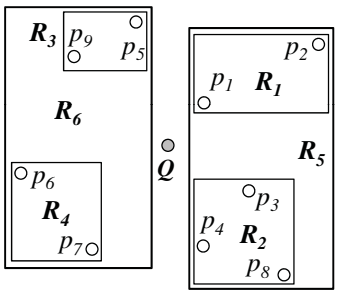
Formally, let the function $dist(\cdot,\cdot)$ denote the Euclidean distance between its argument locations, and let $\mathcal{D}(q.\psi) = \{p \in \mathcal{D} \mid q.\psi \subseteq p.\psi\}$ be the objects in $\mathcal{D}$ that contain all the keywords in q. The result of the *top-k spatial keyword query* q, $q(\mathcal{D})$, is a subset of $\mathcal{D}(q.\psi)$ containing k objects such that $\forall p \in q(\mathcal{D})\ (\forall p' \in \mathcal{D}(q.\psi) - q(\mathcal{D})\ (dist(q.\lambda, p.\lambda) \leq dist(q.\lambda, p'.\lambda)))$. The *joint top-$k$ spatial keyword query* Q is a set $\{q_i\}$ of such queries.

We introduce the following notion to capture useful information on a joint query Q: (i) $Q.\lambda = \mathrm{MBR}_{q_i \in Q}\ q_i.\lambda$ is the minimum bounding rectangle (MBR) of the locations of the subqueries in Q, (ii) $Q.\psi = \cup_{q_i \in Q}\ q_i.\psi$ is the union of the keyword sets of the subqueries in Q, and (iii) $Q.m = \min_{q_i \in Q} |q_i.\psi|$ is the smallest keyword set size of a subquery in Q.

We later define a variable $q_i.\tau$ that captures the upper bound k^{th} nearest neighbor distance of subquery q_i. The value $Q.\tau = \max_{q_i \in Q}\ q_i.\tau$ then represents the maximum upper bound k^{th} nearest neighbor distance of all the subqueries in Q.

Referring to Figure 1, the joint query Q contains three subqueries q_1, q_2, and q_3 (shown as shaded dots). The objects (e.g., restaurants) are shown as white dots. We have that: $q_1.\psi = \{\text{curry,sushi}\}$, $q_2.\psi = \{\text{seafood, sushi}\}$, $q_3.\psi = \{\text{curry, seafood}\}$. Note that $Q.\lambda$ denotes the MBR of the shaded dots in the figure. We also have: $Q.\psi = \{\text{curry, seafood, sushi}\}$ and $Q.m = 2$.

The paper addresses the problem of developing efficient solutions for the processing of the joint top-k spatial keyword query Q. The underlying data is assumed to contain not only a large number of locations, but also a large number of keyword sets. Due to data size and storage cost considerations, the data is stored on disk.



(a) object locations

object	words
p_1	a, b
p_2	a, c
p_3	a, d
p_4	e, f
p_5	a, b
p_6	d, e
p_7	e, f
p_8	d, f
p_9	a, d

(b) object descriptions

Fig. 2. A Dataset of Spatial Keyword Objects

3 Proposed Algorithms

We present the existing IR-tree in Section 3.1 and then proceed to develop a basic and an advanced algorithm, in Sections 3.2 and 3.3, respectively, for processing joint top-k spatial keyword queries. The algorithms are generic and are not tied to a particular index.

3.1 Preliminaries: the IR-Tree

The IR-tree [2], which we use as a baseline, is essentially an R-tree [5] extended with inverted files [32]. The IR-tree's leaf nodes contain entries of the form $(p, p.\lambda, p.di)$, where p refers to an object in dataset $\mathcal{D}$, $p.\lambda$ is the bounding rectangle of p, and $p.di$ is the identifier of the description of p. Each leaf node also contains a pointer to an inverted file with the text descriptions of the objects stored in the node.

An inverted file index has two main components.

- A vocabulary of all distinct words appearing in the description of an object.
- A posting list for each word t that is a sequence of identifiers of the objects whose descriptions contain t.

Each non-leaf node R in the IR-tree contains a number of entries of the form $(cp, rect, cp.di)$ where cp is the address of a child node of R, $rect$ is the MBR of all rectangles in entries of the child node, and $cp.di$ is the identifier of a pseudo text description that is the union of all text descriptions in the entries of the child node.

As an example, Figure 2a contains 8 spatial objects $p_1, p_2, \ldots, p_8$, and Figure 2b shows the words appearing in the description of each object. Figure 3a illustrates the corresponding IR-tree, and Figure 3b shows the contents of the inverted files associated with the nodes.

3.2 Basic Algorithm: IterAte

The IterAte algorithm (Algorithm 1) for computing the joint top-k spatial keyword query is adapted from an existing algorithm [2] that considers a single query.

Recall that a joint top-k spatial keyword query Q consists of a set of subqueries q_i. The IterAte algorithm computes the top-k results for each subquery separately. The arguments are a joint query Q, the root of an index $root$, and the number of results k for each subquery.

When processing a subquery $q_i \in Q$, the algorithm maintains a priority queue U on the nodes to be visited. The key of an element $e \in U$ is the minimum distance $mindist(q_i.\lambda, e.\lambda)$ between the query q_i and the element e. The algorithm utilizes the keyword information to prune the search space. It only loads the posting lists of the words in q_i. A non-leaf entry is pruned if it does not match all the keywords of q_i. The algorithm returns k elements that have the smallest Euclidean distance to the query and contain the query keywords.

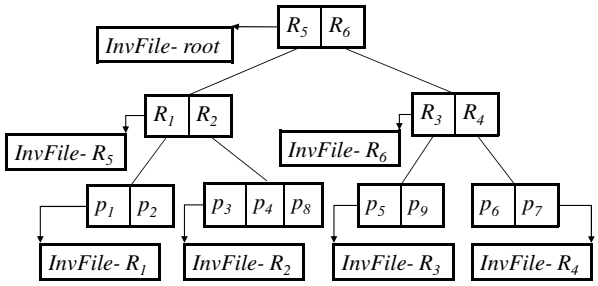
Algorithm 1 IterAte (Joint query Q, Tree root $root$, Integer k)

```
1: for each subquery q_i do
2:     V_i ← new max-priority queue;            ▷ maintain the top k objects
3:     Initialize V_i with k null objects with distance ∞;
4:     U ← new min-priority queue;
5:     U.Enqueue(root, 0);
6:     while U is not empty do
7:         e ← U.Dequeue();
8:         if e is an object then
9:             update V_i by (e, dist(q_i.λ, e.λ));
10:            if V_i has k non-null objects then
11:                break the while-loop;
12:        else                                  ▷ e points to a child node
13:            read the node CN of e;
14:            read the posting lists of CN for keywords in q_i.ψ;
15:            for each entry e' in the node CN do
16:                if q_i.ψ ⊆ e'.ψ then
17:                    U.Enqueue(e', mindist(q_i.λ, e'.λ));
18: return {V_i};                              ▷ top-k results of each subquery
```

Example 1: Consider the joint query $Q = \{q_1, q_2, q_3\}$ in Figure 2, where $q_1.\psi = \{a, b\}$, $q_2.\psi = \{b, c\}$, $q_3.\psi = \{a, c\}$, and all subqueries (and Q) have the same location λ. Table 1 shows the minimum distances between Q and each object and bounding rectangle in the tree.

We want to find the top-1 object. For each subquery q_i, IterAte thus computes the top-1 result. Subquery $q_1 = \langle \lambda, \{a, b\} \rangle$ first visits the root and loads the posting lists of words a and b in $InvFile\text{-}root$. Since entries R_5 and R_6 both contain a and b, both entries are inserted into the priority queue with their distances to q_1. The next dequeued entry is R_5, and the posting lists of words a and b in $InvFile\text{-}R_5$ are loaded. Since only R_1 contains a and b, R_1 is inserted into the queue, while R_2 is pruned.

Now R_6 and R_1 are in the queue, and R_6 is dequeued. After loading the posting lists of words a and b in $InvFile\text{-}R_6$, R_3 is inserted into the queue, while R_4 is pruned. Now R_1 and R_3 are in the queue, and R_1 is dequeued. Its child node is loaded, and the top-1 object p_1 is found, since the distance



(a) IR-Tree

InvF-root	*InvF-R_5*	*InvF-R_6*	*InvF-R_1*	*InvF-R_2*	*InvF-R_3*	*InvF-R_4*
a: R_5, R_6	a: R_1, R_2	a: R_3	a: p_1, p_2	a: p_3	a: p_5,p_9	d: p_6
b: R_5, R_6	b: R_1	b: R_3	b: p_1	d: p_3, p_8	b: p_5	e: p_6, p_7
c: R_5	c: R_1	d: R_3,R_4	c: p_2	e: p_4	d: p_9	f: p_7
d: R_5, R_6	d: R_2	e: R_4		f: p_4, p_8		
e: R_5, R_6	e: R_2	f: R_4				
f: R_5, R_6	f: R_2					

(b) Content of Inverted Files

Fig. 3. Example IR-Tree

of p_1 is smaller than that of the first entry (R_3) in the queue ($2 < 4$). Similarly, the result of subquery $\langle \lambda, \{b, c\} \rangle$ is empty, and the result of subquery $\langle \lambda, \{a, c\} \rangle$ is p_2. □

The disadvantage of the ITERATE algorithm is that it may visit a tree node multiple times, leading to high I/O cost.

TABLE 1
Distances From Q to Objects/Rectangles in Figure 2

Objects	Dist.	Rectangles	Dist.
p_1	2	R_1	2
p_2	5	R_2	2
p_3	6	R_3	4
p_4	7	R_4	5
p_5	3	R_5	0.5
p_6	9	R_6	1
p_7	8	R_7	0
p_8	8		
p_9	3		

3.3 The GROUP Algorithm

The GROUP algorithm aims to process all subqueries of Q concurrently by employing a shared priority queue U to organize the visits to the tree nodes that can contribute to closer results (for some subquery). Unlike ITERATE, GROUP guarantees that each node in the tree is accessed at most once during query processing.

Pruning Strategies.
The algorithm uses three pruning rules. Let e be an entry in a non-leaf tree node. We utilize the MBR and keyword set of e to decide whether its subtree may contain only objects that are farther from or irrelevant to all subqueries of Q.

Pruning Rule 1 prunes a non-leaf entry whose subtree contains objects that have too few relevant keywords for subqueries of Q. This rule is effective when the union keyword set $Q.\psi$ of Q is small, e.g., when many keywords are shared among subqueries of Q.

Pruning Rule 1: Cardinality-Based Pruning.
Let e be an entry in a non-leaf node. If $|(\cup_{q_i \in Q}\ q_i.\psi) \cap e.\psi| < \min_{q_i \in Q} |q_i.\psi|$ then no object in the subtree of e can become a result. Note that the premise is equivalent to $|Q.\psi \cap e.\psi| < Q.m$.

Proof: Let p be any object in the subtree of non-leaf entry e. For any subquery q' of Q, we obtain: $|(\cup_{q_i \in Q}\ q_i.\psi) \cap e.\psi| \geq |q'.\psi \cap e.\psi|$. By the property of the IBR-tree, we have: $p.\psi \subseteq e.\psi$. Therefore, we derive: $|q'.\psi \cap e.\psi| \geq |q'.\psi \cap p.\psi|$. Also, we have: $|q'.\psi| \geq \min_{q_i \in Q} |q_i.\psi|$. Combining the above three inequalities together with the given condition $\min_{q_i \in Q} |q_i.\psi| > |(\cup_{q_i \in Q}\ q_i.\psi) \cap e.\psi|$, we obtain: $|q'.\psi| > |q'.\psi \cap p.\psi|$. Thus, p does not contain all keywords of $q'.\psi$ and cannot become a result of q'. □

Next, Pruning Rule 2 prunes a non-leaf entry whose subtree contains objects that are located too far away from subqueries of Q. This rule is effective when the MBR $Q.\lambda$ is small, i.e., when the subqueries of Q are located within a small region.

Pruning Rule 2: MBR-Based Pruning.
Let e be an entry in a non-leaf node, and let $q_i.\tau$ be an upper bound on the kNN distance of subquery q_i. If $mindist((\mathrm{MBR}_{q_i \in Q} q_i.\lambda), e.\lambda) \geq \max_{q_i \in Q} q_i.\tau$ then no object in the subtree of e can be a result. Note that the premise is equivalent to $mindist(Q.\lambda, e.\lambda) \geq Q.\tau$.

Proof: Let p be any object in the subtree of non-leaf entry e. For any subquery q' of Q, we obtain: $mindist((\mathrm{MBR}_{q_i \in Q} q_i.\lambda), e.\lambda) \leq mindist(q'.\lambda, e.\lambda)$. By the property of the IBR-tree, we derive: $mindist(q'.\lambda, e.\lambda) \leq dist(q'.\lambda, p.\lambda)$. Also, we have: $q'.\tau \leq \max_{q_i \in Q} q_i.\tau$. Combining the above three inequalities together with the given condition $\max_{q_i \in Q} q_i.\tau \leq mindist((\mathrm{MBR}_{q_i \in Q} q_i.\lambda), e.\lambda)$, we obtain: $q'.\tau \leq dist(q'.\lambda, p.\lambda)$. Thus, p cannot become a closer result of q'. □

The CPU time required to check Pruning Rules 1 and 2 is independent of the number of subqueries in Q, as they only need aggregate information about Q. However, the two rules are also loose, as they do not exploit all the specific information in the subqueries in Q.

We adopt the filter-and-refine approach when applying the pruning rules. When neither of the two rules can prune a non-leaf entry e, we apply Pruning Rule 3, which must examine each individual subquery of Q. It first constructs the set Q^* as the subset of subqueries that have the possibility to obtain closer and relevant results from the subtree rooted at e. This subtree only needs to be visited when the set Q^* is non-empty. This rule achieves high pruning power, but at higher computational cost.

Pruning Rule 3: Individual Pruning.
Let e be an entry in a non-leaf node. Let $q_i.\tau$ be an upper bound on the kNN distance of subquery q_i. Let $Q^* = \{q_i \in Q \mid mindist(q_i.\lambda, e.\lambda) \leq q_i.\tau \wedge q_i.\psi \subseteq e.\psi\}$; if $Q^* = \emptyset$, no object in the subtree of e can become a result.

Proof: Let p be any object in the subtree of non-leaf entry e. Let $Q^* = \{q_i \in Q \mid mindist(q_i.\lambda, e.\lambda) \leq q_i.\tau \wedge q_i.\psi \subseteq e.\psi\}$. If Q^* is the empty set then we obtain:

$mindist(q_i.\lambda, e.\lambda) > q_i.\tau \vee q_i.\psi \not\subseteq e.\psi$, for each $q_i \in Q$. By the property of the IBR-tree, we derive: $p.\psi \subseteq e.\psi$, and also $mindist(q_i.\lambda, e.\lambda) \leq dist(q_i.\lambda, p.\lambda)$. Combining the above inequalities together, we obtain: $dist(q_i.\lambda, p.\lambda) > q_i.\tau \vee q_i.\psi \not\subseteq p.\psi$, for each $q_i \in Q$. Thus, p cannot become a closer result of q_i. □

Algorithm.
The GROUP algorithm (Algorithm 2) applies the three pruning rules. The arguments are a joint query Q, the root $root$ of an index, and the number k of results for each subquery.

The priority queue U is first initialized with the root node (lines 1–2). For each subquery $q_i \in Q$, it employs a priority queue V_i to maintain the top-k objects of q_i. The value $q_i.\tau$ denotes the maximum distance in V_i; it is initialized to infinity. The algorithm then executes the while loop in lines 7–30 whenever U is non-empty. The top-k results of each subquery are eventually reported (line 31).

In each iteration, the top entry e (i.e., with the smallest key) is dequeued from U. Specifically, the key of e is defined as its minimum distance to its relevant subqueries of Q. The loop has two phases: (I) checking whether the dequeued entry e can contribute a closer and relevant result for some subquery (lines 8–17), and (II) processing the child node of e (lines 18–30).

Algorithm 2 GROUP (Joint query Q, Tree root $root$, Integer k)

```
 1: U ← new min-priority queue;
 2: U.Enqueue(root, 0);
 3: for each subquery q_i ∈ Q do
 4:     V_i ← new max-priority queue;          ▷ maintain the top k objects
 5:     initialize V_i with k null objects with distance ∞;
 6:     let q_i.τ be the maximum distance in V_i;
 7: while U is not empty do
 8:     e ← U.Dequeue();                       ▷ phase I: checking dequeued entry
 9:     if e.key ≥ Q.τ or |Q.ψ ∩ e.ψ| < Q.m then        ▷ Rules 1, 2
10:         continue the while-loop;
11:     Q* ← {q_i ∈ Q | mindist(q_i.λ, e.λ) ≤ q_i.τ ∧ q_i.ψ ⊆ e.ψ};
12:     if Q* is empty then                    ▷ Rule 3
13:         continue the while-loop;
14:     e.key ← min_{q_i ∈ Q*} mindist(q_i.λ, e.λ);
15:     if e.key has increased (in line 14) then        ▷ update key
16:         U.Enqueue(e, e.key);
17:         continue the while-loop;
18:     read the child node CN of e;   ▷ phase II: processing child node
19:     read the posting lists of CN for keywords in Q*.ψ;
20:     if CN is a non-leaf node then
21:         for each entry e' in the node CN do
22:             if |Q*.ψ ∩ e'.ψ| ≥ Q*.m and mindist(Q*.λ, e'.λ) <
                    Q*.τ then                   ▷ Rules 1,2
23:                 Q' ← {q_i ∈ Q* | mindist(q_i.λ, e'.λ) ≤ q_i.τ ∧
                        q_i.ψ ⊆ e'.ψ };
24:                 if Q' is not empty then     ▷ Rule 3
25:                     U.Enqueue(e', min_{q_i ∈ Q'} mindist(q_i.λ, e'.λ));
26:     else                                   ▷ CN is a leaf node
27:         for each object p in the leaf node CN do
28:             if |Q*.ψ ∩ p.ψ| ≥ Q*.m and mindist(Q*.λ, p.λ) <
                    Q*.τ then                   ▷ Rules 1, 2
29:                 for each subquery q_i ∈ Q* such that
                        dist(q_i.λ, p.λ) < q_i.τ and q_i.ψ ⊆ p.ψ
                        do
30:                     update V_i by (p, dist(q_i.λ, p.λ));
31: return {V_i};                              ▷ top-k results of each subquery
```

In phase I, Rules 1 and 2 are used to check the dequeued entry e. If e cannot be pruned, set Q^* is computed, which contains the subqueries that have the possibility of obtaining closer and relevant results from the subtree of e. Then Rule 3 is applied to check again whether e can be pruned.

Next, we recompute the key $e.key$ of e as its minimum distance to the relevant subqueries in the set Q^* (line 14). It is worth noticing that during the running of the algorithm, the value of $q_i.\tau$ can decrease, leading to a reduction of the set Q^* and thus an increase of $e.key$. In case $e.key$ has increased, we need to insert it into the priority queue U in order to meet the ascending key ordering requirement of U.

If entry e survives, we enter phase II and read the child node CN of e. In order to retrieve (relevant) keywords for entries in CN, the posting lists of CN are read only for keywords in the set $Q^*.\psi$ (line 19). If CN is a non-leaf node, we apply Rules 1, 2, and 3 to each entry e' in CN. Only unpruned entries are enqueued into U. If CN is a leaf node, we apply Rules 1 and 2 to discard non-qualifying objects in CN. The remaining objects are used to update the results for relevant subqueries of Q^*.

We proceed to explain algorithm GROUP with an example.

Example 2: Consider the joint query $Q = \{\langle\lambda, \{a, b\}\rangle, \langle\lambda, \{b, c\}\rangle, \langle\lambda, \{a, c\}\rangle\}$ from Example 1 in Figure 2. Here, $Q.\psi = \{a, b, c\}$ and $Q.m = 2$. All subqueries have the same location λ. We want to find the top-1 object.

The algorithm maintains a queue V_i of size 1 for each subquery to keep track of the current result of each subquery. First, GROUP visits the root of the tree and loads the posting lists of words a, b, and c in $InvFile\text{-}root$. Since $|R_5.\psi| = 3$ ($\geqslant m$; R_5 contains a, b, and c) and $|R_6.\psi| = 2$ ($\geqslant m$; R_6 contains a and b), these two entries are inserted into the priority queue according to their keys (i.e., minimum distances to the subquery locations).

Since the key of R_5 is smaller than that of R_6 ($0.5 < 1$), R_5 is dequeued first. The posting lists of words a, b, and c in $InvFile\text{-}R_5$ are loaded. Since $|R_1.\psi| = 3$ ($\geqslant m$; R_1 contains a, b, and c) and $|R_2.\psi| = 1$ ($< m$; R_2 only contains a), R_1 is inserted into the queue, while R_2 is pruned (Rule 1). Now, there are two entries in the queue: R_6 and R_1.

R_6 is dequeued because of its smaller distance ($1 < 2$). After loading the posting lists of words a, b, and c in $InvFile\text{-}R_6$, since $|R_3.\psi| = 2$ ($\geqslant m$; R_3 contains a and b) and $|R_4.\psi| = 0$ ($< m$; R_4 contains none of a, b, and c), R_3 is inserted into the queue, while R_4 is pruned (Rule 1). Now, R_1 and R_3 are in the queue.

Next, R_1 is dequeued. The top-1 object for subquery $\langle\lambda, \{a, b\}\rangle$ is found, i.e., p_1, and the top-1 object for subquery $\langle\lambda, \{a, c\}\rangle$ is found, i.e., p_2. Then R_3 is dequeued. Since R_3 contains a and b and the distance from R_4 to its closest relevant subquery is 4, which is greater than the result (p_1) of subquery $\langle\lambda, \{a, b\}\rangle$, it is pruned (Rule 2). The queue is empty, and the algorithm terminates and reports the top-k result for each subquery of Q. □

Optimization of the Computational Cost (Bitmap-Opt).
It is expensive to perform lines 11 and 14 in Algorithm 2. A straightforward implementation examines each subquery $q_i \in Q$ once, leading to high computational overhead when Q is large. We attach a bitmap to each en-queued entry. The bitmap length equals the number of subqueries. If an entry is relevant to a subquery, i.e., the entry contains the keywords of the subquery, the corresponding bit of its bitmap is set to

1. During query processing, we then need only examine the relevant subqueries of a dequeued entry, and thus save cost.

4 Proposed Indexes and Adaptations

We present the W-IR-tree (and several variants) that organizes data objects according to both location and keywords in Sections 4.1 and 4.2. We discuss the processing of the joint top-k query using existing indexes in Section 4.3.

4.1 The W-IR-Tree

Word Partitioning.
As a first step in presenting the index structure, we consider the partitioning of a dataset according to keywords. We hypothesize that a keyword query will often contain a frequent word (say w). This inspires us to partition dataset $\mathcal{D}$ into the subset $\mathcal{D}^+$ whose objects contain w and the subset $\mathcal{D}^-$ whose objects do not contain w and that we need not examine when processing a query containing w.

We aim at partitioning $\mathcal{D}$ into multiple groups of objects, such that the groups share as few keywords as possible. However, this problem is equivalent to, e.g., the clustering problem and is NP-hard. Hence, we propose a heuristic to partition the objects.

Let the list W of keywords of objects sorted in descending order of their frequencies be: $w_1, w_2, \ldots, w_m$, where m is the number of words in $\mathcal{D}$. Frequent words are handled before infrequent words. We start by partitioning the objects into two groups using word w_1: the group whose objects contain w_1, and the group whose objects do not. We then partition each of these two groups by word w_2. This way, the dataset can be partitioned into at most $2, 4, \ldots, 2^m$ groups. By construction, the word overlap among groups is small, which will tend to reduce the number of groups accessed when processing a query.

Algorithm 3 recursively applies the above partitioning to construct a list of tree nodes L. To avoid underflow and overflow, each node in L must contain between $B/2$ and B objects, where B is the node capacity. In the algorithm, $\mathcal{D}$ is the dataset being examined, and W is the corresponding word list sorted in the descending order of frequency.

When the number of objects in $\mathcal{D}$ (i.e., $|\mathcal{D}|$) is between $B/2$ and B, $\mathcal{D}$ is added as a node to the result list L (lines 1–2). If $|\mathcal{D}| < B/2$ then $\mathcal{D}$ is returned to the parent algorithm call (lines 3–4) for further processing. If $|\mathcal{D}| > B$ (line 5) then we partition $\mathcal{D}$ (lines 6–19).

In case W is empty (line 6), all the objects in $\mathcal{D}$ must have the same set of words and cannot be partitioned further by words. We hence use a main-memory R-tree with fanout (node capacity) B to partition $\mathcal{D}$ according to location and add the leaf nodes of the R-tree to the result list L (lines 7–8).

When W is non-empty, we take the most frequent (first) word in W (lines 9–10). The objects in $\mathcal{D}$ are partitioned into groups $\mathcal{D}^+$ and $\mathcal{D}^-$ based on whether or not they contain w (lines 11–12). Next, we recursively partition $\mathcal{D}^+$ and $\mathcal{D}^-$ (lines 13–14). The remaining objects from these recursive calls (i.e., the sets T^+ and T^-) are then merged into the set T' (line 15). If T' has enough objects, it is added as a node to L (lines 16–17). Otherwise, set T' is returned to the parent algorithm call (lines 18–19).

If the initial call of the algorithm returns a group with less than $B/2$ objects, it is added as a node to the result list L, since no more objects are left to be merged.

Algorithm 3 WordPartition (Dataset $\mathcal{D}$, Sorted list of words W, Integer B, List of tree nodes L)

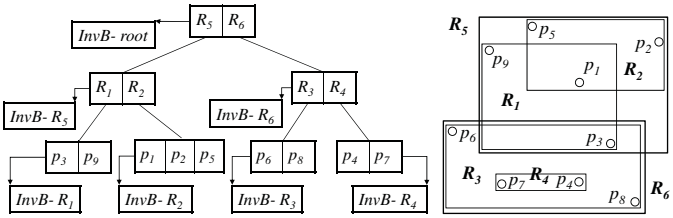
```
1: if B/2 ≤ |D| ≤ B then
2:     add D as a node to L;
3: else if |D| < B/2 then
4:     return D;
5: else                                              ▷ partitioning phase
6:     if W is empty then                            ▷ partitioning by location
7:         insert D into a main-memory R-tree with fanout B;
8:         add the leaf nodes of the main-memory R-tree to L;
9:     else                                          ▷ partitioning by words
10:        w ← first word in W; W ← W \ {w};
11:        D+ ← {p ∈ D | w ∈ p.ψ};
12:        D− ← {p ∈ D | w ∉ p.ψ};
13:        T+ ← WordPartition(D+, W, B, L);
14:        T− ← WordPartition(D−, W, B, L);
15:        T' ← T+ ∪ T−;
16:        if B/2 ≤ |T'| ≤ B then
17:            add T' as a node to L;
18:        else if |T'| < B/2 then
19:            return T';
```

Example 3: Consider the 8 objects in Figure 2b and let the node capacity B be 3. Words are sorted by their frequencies in the dataset, and ties are broken according to alphabetic order. Thus, we obtain the list: $W = \langle (a,5), (d,4), (f,3), (b,2), (e,2), (c,1) \rangle$. Using word a, the objects are partitioned into the groups $\mathcal{D}^+ = \{p_1, p_2, p_3, p_5, p_9\}$ and $\mathcal{D}^- = \{p_4, p_6, p_7, p_8\}$. Both contain more than 3 objects and are partitioned according to word d, resulting in $\mathcal{D}^{++} = \{p_1, p_2, p_5\}$, $\mathcal{D}^{+-} = \{p_3, p_9\}$, $\mathcal{D}^{-+} = \{p_6, p_8\}$, and $\mathcal{D}^{--} = \{p_4, p_7\}$. All groups but $\mathcal{D}^{+-}$ are added to result list L, as they contain between 1.5 and 3 objects. Group $\mathcal{D}^{+-}$ is passed to the first call of the algorithm and is finally added to L. □

Tree Construction Using Word Partitioning.
The W-IR-tree uses the same data structures as the IR-tree, but is constructed differently by using word partitioning (thus the prefix 'W'). Instead of performing insertions iteratively, we build the W-IR-tree bottom-up.

We first use the word partitioning (Algorithm 3) to obtain the groups that will form the leaf nodes of the W-IR-tree. For each leaf node N, we compute $N.\psi$ as the union of the words of the objects in node N, and $N.\lambda$ as the MBR of the objects in N. Next, we regard the leaf nodes as objects and apply Algorithm 3 to partition the leaf nodes into groups that form the nodes at the next level in the W-IR-tree. We repeat this process until a single W-IR-tree root node is obtained. Figure 4a illustrates the W-IR-tree for the 8 objects in Figure 4b. Following Example 3, leaf nodes $R_1 \leftarrow \{p_3, p_9\}$, $R_2 \leftarrow \{p_1, p_2, p_5\}$, $R_3 \leftarrow \{p_6, p_8\}$, and $R_4 \leftarrow \{p_4, p_7\}$ are first formed. Figure 4b shows the MBRs of those leaf nodes and $R_1.\psi = \{a, d\}$, $R_2.\psi = \{a, b, c\}$, $R_3.\psi = \{d, e, f\}$, $R_4.\psi = \{e, f\}$. Next, Algorithm 3 is used to partition R_1, R_2, R_3, and R_4, since they are 4 (the node capacity is 3) nodes and cannot be put into one node at the next level. Using

word a, two partitions are obtained, i.e., $R_5 \leftarrow \{R_1, R_2\}$ and $R_6 \leftarrow \{R_3, R_4\}$. Since R_5 and R_6 contain between 1.5 and 3 nodes, there is no need to further partition them. Finally, R_5 and R_6 can be put into one node at the next level, resulting the root node of the W-IR-tree.



(a) tree structure (b) object locations

Fig. 4. Example W-IR-Tree

The IR-tree and the W-IR-tree organize the data objects differently. The IR-tree organizes the objects purely based on their spatial locations. In contrast, the W-IR-tree first partitions the data objects based on their keywords (Lines 10–19) and then further partitions them based on their spatial locations (Lines 6–8). Thus, the W-IR-tree matches better the semantics of the top-k spatial keyword query and has the potential to perform better than the IR-tree.

Updates.
A deletion in the W-IR-tree is done as in the R-tree, by finding the leaf node containing the object and then removing the object. An insertion selects a branch such that the insertion of the object leads to the smallest "enlargement" of the keyword set, meaning that the number of distinct words included in the branch increases the least if the object is inserted there.

4.2 The W-IBR-Tree and Variants

Inverted Bitmap Optimization.
Each node in the W-IR-tree contains a pointer to its corresponding inverted file. By replacing each such inverted file by an inverted bitmap, we can reduce the storage space of the W-IR-tree and also save I/O during query processing. We call the resulting tree the W-IBR-tree.

Table 2 illustrates the inverted bitmaps that correspond to the nodes of the W-IBR-tree in Figure 4. A bitmap position corresponds to the relative position of an entry in its W-IBR-tree node. The length of a bitmap is equal to the fanout of a node. For example, the node R_3 stores the points p_6 and p_8. The inverted list for item f of R_3 is p_8 (which is the second entry in R_3). Thus, the inverted bitmap for item f of R_3 is '01'.

TABLE 2
Content of Inverted Bitmaps of the W-IBR-Tree

InvB-root	R_5	R_6	R_1	R_2	R_3	R_4
a: 10	a: 11	c: 01	a: 11	a: 111	d: 11	c: 11
b: 10	b: 01	d: 10	d: 11	b: 101	e: 10	f: 11
c: 11	c: 01	e: 10		c: 010	f: 01	
d: 11	d: 10	f: 11				
e: 01						
f: 01						

IBR-Tree and CD-IBR-Tree.
The inverted bitmap optimization technique is applicable to the original IR-tree [2], yielding the IBR-tree. The bitmap optimization technique is also applicable to the CDIR-tree [2], yielding the CD-IBR-tree.

4.3 Query Processing Using Other Indexes

The ITERATE and GROUP algorithms are generic and are readily applicable to the proposed indexes, including the W-IR-tree and the W-IBR-tree; the existing indexes, e.g., the IR-tree and the CDIR-tree; and the existing indexes using the optimizations proposed in this paper, including the IBR-tree and CD-IBR-tree. More detailed information is covered in Appendix A and B.

5 I/O Cost Models for the IR-Tree and the W-IR-Tree

A cost model for an index is affected by how the objects are organized in the index. The Inverted-R-trees, the IR^2-tree, the IR-tree, and the IBR-tree adopt spatial proximity, as does the R-tree, to group objects into nodes. The W-IR-tree and the W-IBR-tree use word partitioning to organize objects. This section develops an I/O cost model for each of the IR-tree and the W-IR-tree that then serve as representatives of the above two index families. The cost model of the CD-IBR-tree is in-between those of the two families, since it considers both spatial proximity and text relevancy.

Specifically, the models aims to capture the number of leaf node accesses, assuming that the memory buffer is large enough for the caching of non-leaf nodes.[2] We also ignore the I/O cost of accessing posting lists as this cost is proportional to the I/O cost of accessing the tree. The resulting models provide insight into the performance of the indexes.

Like previous work on R-tree cost modeling [24], we make certain assumptions to render the cost models tractable. We assume that the locations of objects and queries are uniformly distributed in the unit square $[0, 1]^2$. Let n be the number of objects in the dataset $\mathcal{D}$, and let B be the average capacity of a tree node. Thus, the number of leaf nodes is $N_L = n/B$.

We assume that the frequency of keywords in the dataset follows a Zipfian distribution [16], [31], which is commonly observed from the words contained in documents. Let the word w_i be the i-th most frequent word in the dataset. The occurrence probability of w_i is then defined as

$$F(w_i) = \frac{\frac{1}{i^s}}{\sum_{j=1}^{N_w} \frac{1}{w_j^s}}, \tag{1}$$

where N_w is the total number of words and s is the value of the exponent characterizing the distribution (skew).

Let a query be $q = \langle \lambda, \psi \rangle$. Suppose that each object and the query contain z keywords. We assume that the words of each object are drawn *without replacement* based on the occurrence probabilities of the words. Let d_{knn} denote the kNN distance of q, i.e., the distance to the k^{th} nearest neighbor in $\mathcal{D}$. Let e

2. With a typical node capacity in the hundreds and a fill-factor of approximately 0.7, the leaf level makes up well beyond 99% of the index.

be any non-leaf entry that points to a leaf node. We need to access e's leaf node when:

1) the keyword set of e contains $q.\psi$, and
2) the minimum distance from q to e is within d_{knn}.

Let the probability of the above two events be the *keyword containment probability* $Pr(e.\psi \supseteq q.\psi)$ and the *spatial intersection probability* $Pr(mindist(q, e.\lambda) \leq d_{knn})$, respectively. Thus, the access probability of the child node of e is:

$$Pr(\text{access } e) = Pr(e.\psi \supseteq q.\psi) \cdot Pr(mindist(q, e.\lambda) \leq d_{knn}).$$

We then estimate the total number of accessed leaf nodes as:

$$\mathcal{COST} = N_L \cdot Pr(\text{access } e) \tag{2}$$

We proceed to derive the probability of an object matching the query keywords and the kNN distance d_{knn}. We then study the probabilities $Pr(e.\psi \supseteq q.\psi)$ and $Pr(mindist(q, e.\lambda) \leq d_{knn})$ for the IR-tree and the W-IR-tree, respectively. Finally, we compare the two trees using the cost models.

5.1 Estimation of Keyword Probability and kNN Distance

Probability of an Object Matching the Query Keywords. Let $F(q.\lambda)$ be the probability of having $q.\lambda$ as the keyword set of an object of $\mathcal{D}$. Let z be the number of words in each object (and also in the query). Let an arbitrary list (i.e., sequence) of $q.\lambda$ be $w_{q_1}, w_{q_2}, \cdots, w_{q_z}$. Due to the "without replacement" rule, when we draw the j-th word of an object, any previously drawn word $(w_{q_1}, w_{q_2}, \cdots, w_{q_{j-1}})$ cannot be drawn again. Thus, the probability of the j-th drawn word being w_{q_j} is:

$$\frac{F(w_{q_j})}{Pr(w_{q_j} \mid w_{q_j} \notin \cup_{h=1}^{j-1}\{w_{q_h}\})} = \frac{F(w_{q_j})}{1 - \sum_{h=1}^{j-1} F(w_{q_h})}.$$

Then the probability of drawing the exact list w_{q_1}, w_{q_2}, $\cdots$, w_{q_z} is:

$$\prod_{j=1}^{z} \frac{F(w_{q_j})}{1 - \sum_{h=1}^{j-1} F(w_{q_h})}.$$

We then sum up the probability for every list enumeration of $q.\lambda$. As there are $z!$ such lists, the probability of having $q.\lambda$ as the keyword set of an object $p \in \mathcal{D}$ is:

$$\begin{aligned} F(q.\lambda) &= \sum_{\text{any list of } q.\lambda} \prod_{j=1}^{z} \frac{F(w_{q_j})}{Pr(w_{q_j} \mid w_{q_j} \notin \cup_{h=1}^{j-1}\{w_{q_h}\})} \\ &= \sum_{\text{any list of } q.\lambda} \prod_{j=1}^{z} \frac{F(w_{q_j})}{1 - \sum_{h=1}^{j-1} F(w_{q_h})} \\ &\approx z! \cdot \frac{\prod_{j=1}^{z} F(w_{q_j})}{\left(1 - \frac{\sum_{h=1}^{z} F(w_{q_h})}{2z}\right)^{z-1}} \end{aligned}$$

kNN Distance.
Observe that the kNN distance d_{knn} of q depends only on the dataset $\mathcal{D}$, but is independent of the tree structure.

The number of objects having the keyword $q.\psi$ is: $n \cdot F(q.\psi)$. By substituting this quantity into the estimation model of Tao et al. [23], we estimate the kNN distance of q as

$$d_{knn} = \frac{2}{\sqrt{\pi}} \cdot \left(1 - \sqrt{1 - \sqrt{\frac{k}{n \cdot F(q.\psi)}}}\right).$$

We then approximate the kNN circular region $\odot(q, d_{knn})$ by a square having the same area, i.e., with the side-length $l = \sqrt{\pi} \cdot d_{knn}$. According to Theodoridis et al. [24], the spatial intersection probability is:

$$Pr(mindist(q, e.\lambda) \leq d_{knn}) \approx (\sigma + l)^2 = (\sigma + \sqrt{\pi} \cdot d_{knn})^2, \tag{3}$$

where σ is the side-length of non-leaf entry e. We shortly provide detailed estimates of σ for both trees.

5.2 I/O Cost Models

IR-Tree.
The construction of the IR-tree proceeds as for the R-tree [33]. Objects are partitioned into leaf nodes based on their spatial proximity. We consider the standard scenario where the leaf nodes are squares and form a disjoint partitioning of the unit square. Therefore, we have $N_L \cdot \sigma_{ir}^2 = 1$, and we then obtain $\sigma_{ir} = \sqrt{1/N_L}$. The spatial intersection probability is derived by substituting σ_{ir} into Equation 3.

Given the query keyword $q.\psi$, the keyword set of e contains $q.\psi$ if some object (in the child node) of e has the keyword $q.\psi$. Note that in the IR-tree, the keywords of the objects in the leaf node pointed to by e are distributed randomly and independently. Since the leaf node has capacity B, the keyword containment probability is $Pr_{ir}(e.\psi \supseteq w) = 1 - (1 - F(q.\psi))^B$.

W-IR-Tree.
During the construction of the W-IR-tree, the objects are first partitioned based on their keywords. Thus, the number of leaf nodes that contain the query word $q.\psi$ is: $\max\{N_L \cdot F(q.\psi), 1\}$. Then the objects in these nodes are further partitioned into nodes based on their locations. By replacing N_L with $\max\{N_L \cdot F(q.\psi), 1\}$ in Equation 3, we obtain

$$\sigma_{wir} = \sqrt{1/\max\{N_L \cdot F(q.\psi), 1\}}. \tag{4}$$

We obtain the spatial intersection probability by substituting σ_{wir} into Equation 3.

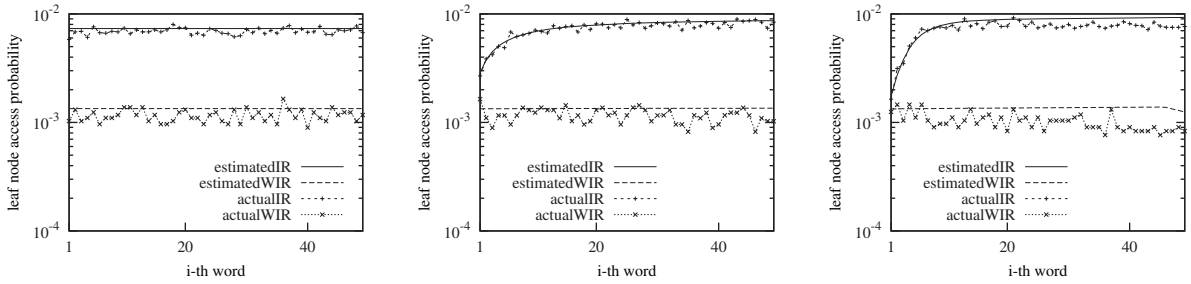
Observe that out of N_L leaf nodes, $\max\{N_L \cdot F(q.\psi), 1\}$ nodes contain the query keyword $q.\psi$. Thus, the keyword containment probability is

$$Pr_{wir}(e.\psi \supseteq q.\psi) = \max\left\{F(q.\psi), \frac{1}{N_L}\right\}. \tag{5}$$

5.3 Theoretical and Empirical Comparisons

Comparisons and Simulation Results.
Comparing the above equations, it holds that σ_{wir} is usually larger than σ_{ir}. In contrast, $Pr_{wir}(e.\psi \supseteq q.\psi)$ is much smaller than $Pr_{ir}(e.\psi \supseteq q.\psi)$. Note that the value $Pr_{ir}(e.\psi \supseteq q.\psi)$ is close to 1 at a typical node capacity B (i.e., in the hundreds).

To illustrate the cost models, we consider an example with $n = 100,000$, $k = 10$, and $z = 1$. For the trees, the default



(a) $s = 0$ (b) $s = 0.75$ (c) $s = 1.5$

Fig. 5. Leaf Node Access Probabilities in the Cost Models, $n = 100,000$, $k = 10$, $z = 1$

maximum capacity is 100 and the fill factor is 0.7, so the average capacity is $B = 100 \cdot 0.7 = 70$. Figure 5 plots the leaf node access probability as a function of the query keyword w_i for three different values of s. The larger the value of s, the more skewed the word distribution becomes ($s = 0$ means the words are uniformly distributed). The estimatedIR and estimatedWIR curve are derived from Equation 2. The actualIR and actualWIR curve are obtained experimentally from synthetic data. The estimated probabilities are close to the actual probabilities. Recall that w_i denotes the i-th most frequent word in the dataset. Since the IR-tree groups objects solely based on location and the W-IR-tree groups objects based on keywords, the W-IR-tree outperforms the IR-tree for all types of query words in terms of leaf node access probability.

Theoretical Comparison for the Uniform Keyword Case ($s = 0$).

We next give a detailed theoretical comparison between the leaf node access probabilities of the IR-tree and the W-IR-tree. For the sake of simplicity, we consider $s = 0$, $z = 1$, and $k = 1$. We derive $l = \sqrt{\pi} \cdot d_{knn} \approx \sqrt{N_w/n}$. Equation 1 is simplified to $1/N_w$. When n is arbitrarily large (as is N_L), Equation 4 and 5 are simplified to $\sqrt{N_w/N_L}$ and $1/N_w$. The leaf node access probability of the W-IR-tree is:

$$Pr_{wir} = \frac{(\sqrt{N_w} + \sqrt{B \cdot N_w})^2}{n \cdot N_w}. \quad (6)$$

The leaf node access probability of the IR-tree is:

$$Pr_{ir} = (\frac{1}{N_L} + l^2 + \frac{2 \cdot l}{\sqrt{N_L}}) \cdot \frac{N_w^B - (N_w - 1)^B}{N_w^B}. \quad (7)$$

Since $(N_w - 1)^B$ is dominated by $N_w^B - B \cdot N_w^{B-1}$, we have:

$$Pr_{ir} \approx \frac{B \cdot (\sqrt{N_w} + \sqrt{B})^2}{n \cdot N_w}. \quad (8)$$

We proceed to identify the conditions when the W-IR-tree or the IR-tree achieves the better performance. By solving $Pr_{wir} \leq Pr_{ir}$, we obtain the condition $N_w \leq B^2$. Similarly, by solving $Pr_{wir} > Pr_{ir}$, we get the condition $N_w > B^2$.

Next, we study the performance gap between the W-IR-tree and the IR-tree in these two cases. Let $N_w = t \cdot B^2$.

When $t > 1$, indicating the W-IR-tree is worse than the IR-tree ($Pr_{wir} \geq Pr_{ir}$), we compute the limit of the difference:

$$\lim_{t \to \infty} Pr_{wir} - Pr_{ir} = \frac{(1 - \frac{1}{t}) + 2 \cdot \sqrt{B} \cdot (1 - \frac{1}{\sqrt{t}})}{n} = \frac{1 + 2 \cdot \sqrt{B}}{n} \leq \frac{2}{n} + \frac{1}{N_L}.$$

Since n is arbitrarily large and $1/N_L$ refers to the probability of accessing one leaf node, when $N_w > B^2$, the W-IR-tree is only slightly worse than the IR-tree.

When $t < 1$, since $N_w \geq 1$, the smallest possible value of t is $1/B^2$, and we compute the limit of the difference as follows:

$$\lim_{t \to \frac{1}{B^2}} Pr_{ir} - Pr_{wir} = \frac{\frac{1}{t} + 2 \cdot \sqrt{B} \cdot \sqrt{\frac{1}{t}} - (2 \cdot \sqrt{B} + 1)}{n} = \frac{(\sqrt{B} + 1)^2 \cdot (B - 1)}{n} \approx \frac{B^2}{n} = \frac{B}{N_L}.$$

So the IR-tree can access B times more leaf nodes than does the W-IR-tree.

In summary, the analysis suggests that the W-IR-tree is significantly better than the IR-tree when $t < 1$ and only slightly worse than the IR-tree when $t > 1$ for uniformly distributed words.

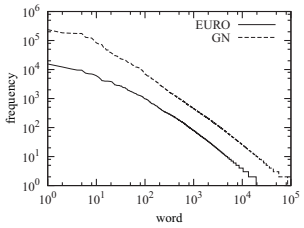
6 Experimental Study

6.1 Experimental Setup

We use two real datasets for studying the robustness and performance of the different approaches. Dataset "EURO" contains points of interest (e.g., pubs, banks, cinemas) in Europe (www.pocketgpsworld.com). Dataset "GN" is obtained from the U.S. Board on Geographic Names (geonames.usgs.gov). Here, each object has a geographic location and a short description. Figure 6(a) offers details on EURO and GN. The spatial domain of each dataset is normalized to the unit square $[0, 1]^2$. Figure 6(b) plots the word frequency distributions in the two datasets. They follow the Zipfian distribution. Few words are generic and frequent, while most words are specific and infrequent [16], [31].

Dataset	EURO	GN
# of objects	162,033	1,868,821
# of distinct words	35,315	222,407
Average # of words per object	18	4

(a) Dataset Details



(b) Word Frequencies

Fig. 6. A Dataset of Spatial Keyword Objects

Regarding the keyword set of a query, we randomly choose an object and then randomly choose words from the object.

Unless stated otherwise, the number w of keywords per query is 3, the number k of results per query is 10, and the number L of subqueries in a joint query is 100.

All index structures are disk resident, and the page size is fixed at 4 KBytes. For all the indexes, the fanout B is 100. All algorithms were implemented in Java and executed on a server with two processors (Intel(R) Xeon(R) Quad-Core CPU E5320 @ 1.86GHz) and 2GB memory.

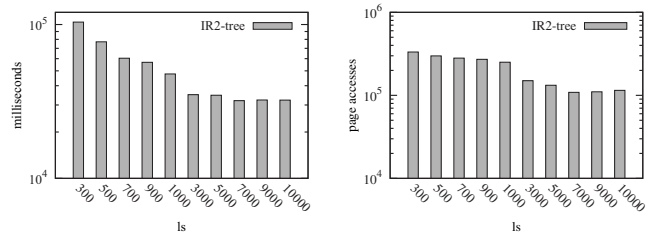
6.2 Tuning Experiments

To favor our competitors, we tune the parameters in the IR^2-tree [3] and the CD-IBR-tree to achieve their best performance on the two datasets.

6.2.1 Tuning the IR^2-Tree

Each entry stores a signature as a fixed-length bitmap that summarizes the text descriptions of objects enclosed in its subtree. The length of the signature l_s affects the performance of the IR^2-tree. Long signatures provides more accurate information than do short signatures, and thus more subtrees can be pruned, incurring less I/O on tree nodes. However, long signatures occupy more space than do short signatures, and thus incur more I/O on signature files. There is a relation between the length of the signatures and performance. We vary the length of the signature l_s to find the best value experimentally. The IR^2-tree with its best l_s is used as one competitor of the proposed solution.

Figure 7 shows the elapsed time and the I/O cost on the EURO dataset when varying l_s from 300 to 10,000. The IR^2-tree performs the best on EURO when $l_s = 7,000$. We have also conducted tuning experiments for the GN dataset, varying l_s from 200 to 200,000. The IR^2-tree performs the best on GN when $l_s = 10,000$.



(a) Elapsed Time (b) I/O

Fig. 7. Varying Signature Length l_s on EURO

6.2.2 Tuning the CD-IBR-Tree

The construction of the CD-IBR-tree consists of two steps, involving two parameters. Step 1 is the building of a DIR-tree with the weight β $(0 \leq \beta \leq 1)$ assigned to spatial distance when inserting an object (and thus $1 - \beta$ is the weight assigned to document similarity). When $\beta = 1$, the DIR-tree is actually an IR-tree. When $\beta = 0$, objects are organized solely according to document similarity. Step 2 is the integration of cluster information of objects into the DIR-tree. In our experiments, we apply the inverted bitmap optimization technique, resulting in the CD-IBR-tree. The number of clusters c is the second parameter. A larger c provides a tighter bound on the word information in a subtree, but needs more space. A smaller c gives a looser bound, but takes up less storage. There is thus a relation between β and the performance, and between c and the performance. Experimentally, we first determine the best β for the DIR-tree and then use the DIR-tree with the best β to determine the best c. The resulting CD-IBR-tree is taken as a competitor of the proposed solution.

Figure 8 shows the elapsed time and the I/O cost on the EURO dataset when varying β from 0 to 1. The DIR-tree performs best on EURO when $\beta = 0.9$. Figure 9 shows the elapsed time and the I/O cost on the EURO dataset when varying c from 4 to 64. As shown, the CD-IBR-tree performs best on EURO when $c = 16$.

We have also conducted tuning experiments for the GN dataset, i.e., varying β from 0 to 1 and varying c from 4 to 64. The DIR-tree performs best on GN when $\beta = 0.9$ and the CD-IBR-tree performs best on GN when $c = 64$.

6.3 Index Statistics

Table 3 shows the sizes of the indexes on the two datasets. The Inverted-R-trees is the largest, since it contains an R-tree for each word and since many objects are indexed multiple times. In the IR^2-tree, the signature length is set to 7,000 and 10,000, and the number of clusters in the CD-IBR-tree is 16 and 64 on EURO and GN dataset, respectively. The W-IBR-tree requires less space than the W-IR-tree for both datasets. Since the CD-IBR-tree incorporates cluster information, it takes more space than does the W-IBR-tree.

TABLE 3
Index Sizes of the EURO and GN Datasets (MB)

Dataset	Inverted-R-trees	IR^2	CD-IBR	W-IR	W-IBR
EURO	495	160	71	108	50
GN	2970	2424	1558	883	422

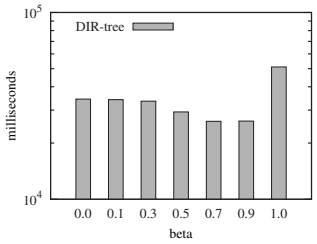
Table 4 shows the average MBR area and the average word size (i.e., number of distinct words) of leaf nodes for these indexes. The W-IBR-tree has the lowest average word size while the IR^2-tree has the smallest average MBR area.

TABLE 4
Leaf-Node Statistics Vs. Index Construction Methods

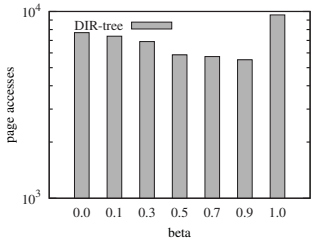
Index	Average MBR area		Average word size	
	EURO	GN	EURO	GN
W-IBR-tree	0.26	0.16	89	50
CD-IBR-tree	0.0056	0.0052	97	71
IR^2-tree	0.00017	0.000013	116	82

6.4 Performance Evaluation

We proceed to consider the performance of the solutions on the EURO and GN datasets. To evaluate the proposed ITERATE and GROUP algorithms, we apply them to the proposed W-IBR-tree and the existing IR^2-tree [3]. The CD-IBR-tree that improves on the existing CDIR-tree [2] by using the techniques proposed in this paper is taken as another competitor. For instance, the solution G-W-IBRtree applies the GROUP algorithm to the W-IBR-tree. We also compare with the specialized algorithm for the Inverted-R-trees.

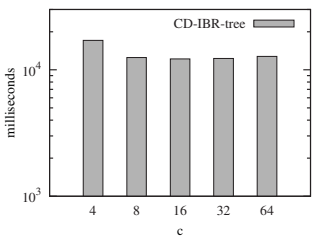


(a) Elapsed Time

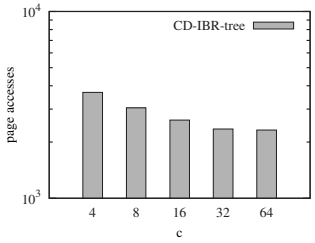


(b) I/O

Fig. 8. Varying β in the DIR-Tree on EURO

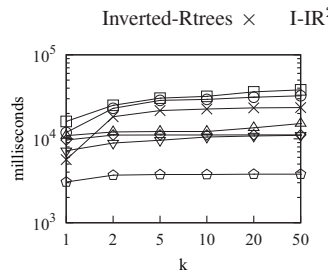


(a) Elapsed Time

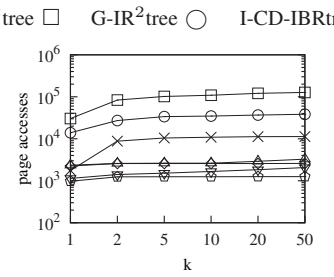


(b) I/O

Fig. 9. Varying c in the CD-IBR-Tree on EURO

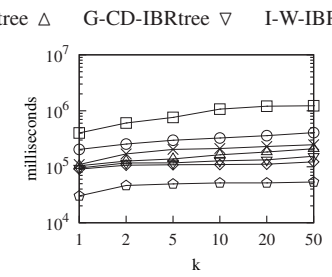


(a) Elapsed Time

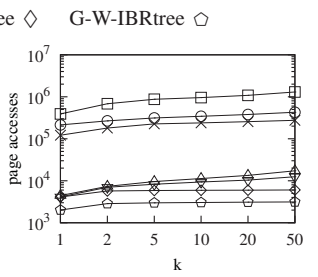


(b) I/O

Fig. 10. Effect of k on Euro

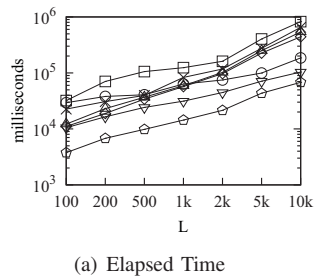


(a) Elapsed Time

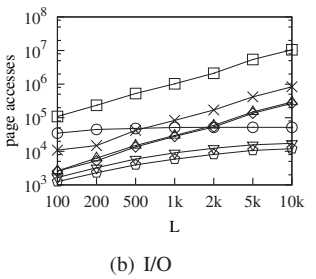


(b) I/O

Fig. 11. Effect of k on GN

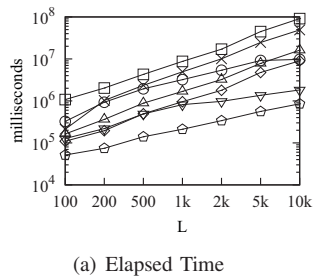


(a) Elapsed Time

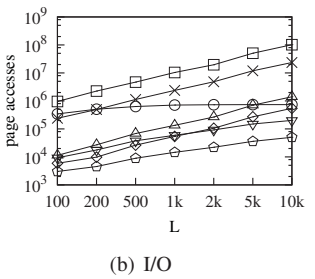


(b) I/O

Fig. 12. Effect of L on Euro

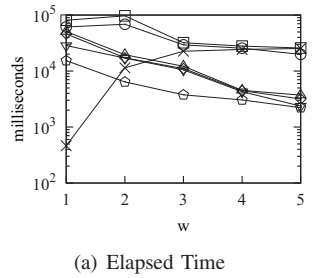


(a) Elapsed Time

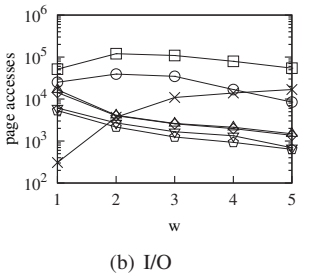


(b) I/O

Fig. 13. Effect of L on GN

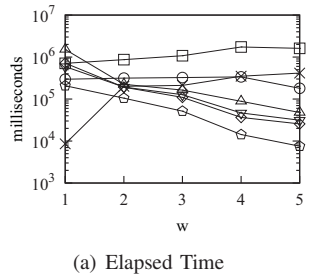


(a) Elapsed Time

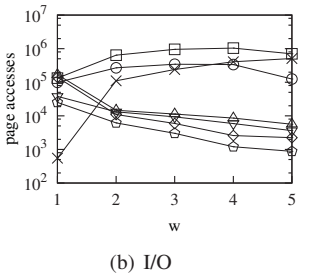


(b) I/O

Fig. 14. Effect of w on Euro



(a) Elapsed Time



(b) I/O

Fig. 15. Effect of w on GN

Effect of k (Number of Results per Subquery). Figures 10 and 11 show the total I/O cost and elapsed time of the solutions as a function of k. As k increases, more tree nodes are accessed, and thus more I/O and elapsed time are incurred. The Inverted-R-trees is not efficient because it accesses many objects that contain part of the query keywords, but not necessarily all the query keywords. In the worst case, it may access all objects in the inverted R-trees to obtain the results. Since the GROUP algorithm processes queries jointly, it exploits opportunities to share disk accesses among subqueries. Such shared disk pages are only visited once. Therefore, the cost of GROUP is lower than that of ITERATE. W-IBR-tree-based solutions significantly outperform non-W-IBR-tree-based solutions, since the nodes of the W-IBR-tree have fewer common words so that the search by a query only involves few branches in the W-IBR-tree, due to the pruning power of the query keywords, using both ITERATE and GROUP. The best solution is GROUP on the W-IBR-tree.

Effect of L (Number of Subqueries in a Joint Query). Figures 12 and 13 show the total I/O cost and elapsed time of the solutions when varying L. As the number of subqueries increases, the costs of Inverted-R-trees and ITERATE increase proportionally because they process subqueries one by one. The cost of GROUP is low since it visits any tree node at most once. The W-IBR-tree outperforms the CD-IBR-tree and the IR^2-tree consistently with both the ITERATE and the GROUP algorithms.

Effect of w (Number of Keywords per Subquery).
Figures 14 and 15 show the total I/O cost and elapsed time of the solutions when varying w. An increasing number of query keywords has two effects: (i) it becomes less likely for a node to contain all query keywords, and (ii) the query results become more distant from the query objects. The former effect may reduce the I/O cost, while the latter may increase the cost.

According to Table 4, the CD-IBR-tree and the IR^2-tree contain leaf nodes with large word sizes and thus cannot benefit much from the first effect. On the other hand, the W-IBR-tree contains leaf nodes with small word sizes, so it is capable of utilizing the first effect. Thus, the W-IBR-tree outperforms the CD-IBR-tree and the IR^2-tree. For the reasons mentioned earlier, GROUP beats ITERATE.

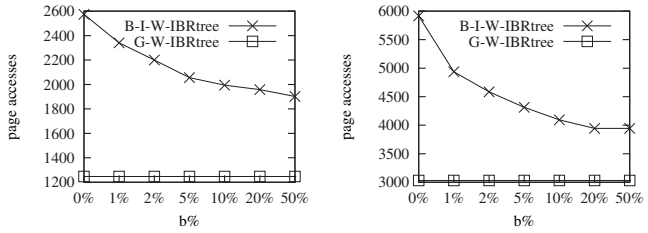
As the number of query keywords increases from 1 to 2, the cost of the Inverted-R-trees increases rapidly because it visits more R-trees. The cost of the Inverted-R-trees increases slowly when the number of query keywords increases from 2 to 5.

Summary.
For each dataset and index, the GROUP algorithm significantly outperforms the ITERATE algorithm in terms of I/O and elapsed time. This is because the GROUP algorithm processes all subqueries jointly and shares the computation among them. In contrast, the ITERATE algorithm processes queries one-at-a-time and may visit some tree node multiple times. Since the signatures in the IR^2-tree only capture keyword information approximately, false positives can occur. Substantial I/O cost and elapsed time may result from visiting irrelevant branches. Unlike the other indexes, the Inverted-R-trees cannot utilize both the spatial and keyword information together to prune the search space. In terms of the construction methods of the trees, the W-IBR-tree achieves better query performance than the CD-IBR-tree and the IR^2-tree. Since the W-IBR-tree has the lowest average word size (i.e., number of distinct words) in leaf nodes (see Table 4), it is effective at pruning early irrelevant nodes that do not contain query keywords. We conclude that the GROUP algorithm with the W-IBR-tree is the best proposal. In the following experiments, we only report I/O costs as these dominate the overall processing costs.

6.5 Buffering and Optimizations

Effect of Buffering.
This experiment uses an LRU main memory buffer for the ITERATE algorithm, denoted by B-ITERATE, and compares it with the GROUP algorithm on the W-IBR-tree. We vary the buffer size $b\%$ from 1% to 50% of the index pages (including both tree nodes and the inverted index) and report the I/O costs on EURO and GN in Figure 16. GROUP outperforms B-ITERATE even when half of the index pages are buffered.

Computational Optimization (Bitmap-Opt) for GROUP.
This experiment evaluates the effectiveness of Bitmap-Opt as proposed for GROUP in Section 3.3. We use *GROUP to denote a variant of GROUP without Bitmap-Opt, and we compare the ITERATE, GROUP, and *GROUP algorithms when used with the W-IBR-tree.



(a) I/O on EURO (b) I/O on GN

Fig. 16. Effect of Buffering

Figures 17 and 18 plot the total elapsed time and the number of comparisons between subqueries and index entries, on the two datasets, as a function of the number of subqueries L. We see that Bitmap-Opt saves considerable numbers of comparisons between subqueries and index entries, thus yielding lower elapsed times when compared to ITERATE and *GROUP. Without the Bitmap-Opt optimization, *GROUP may incur more comparisons than ITERATE. Unlike in ITERATE, the same tree nodes are not visited redundantly in *GROUP. This explains why the elapsed time of *GROUP is less than that of ITERATE. Since Bitmap-Opt enables subqueries to be compared only with relevant index entries, the elapsed time of GROUP grows more slowly than does that of ITERATE.

Effect of Inverted Bitmap Optimization.
In this experiment, we apply the GROUP and ITERATE algorithms on the W-IBR-tree and the W-IR-tree to evaluate the effectiveness of the inverted bitmap optimization. Figure 19 shows the I/O costs of the methods when varying k (the number of results per subquery). The W-IBR-tree accesses compact inverted bitmaps to retrieve keywords of entries/objects, so it incurs lower cost than does the W-IR-tree (using inverted files).

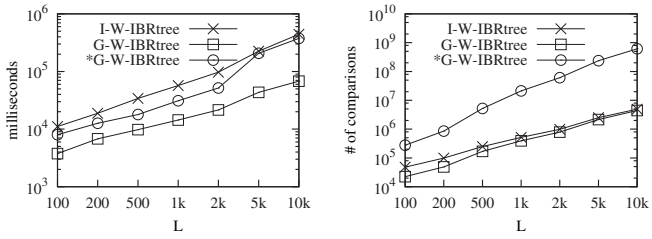
6.6 Extent of Workload Sharing

Effect of Different Types of Keywords.
This experiment studies the performance of the solutions for different types of query keywords:

- Random keywords (R-key), picking a random data object and then random words of the object as the query keyword set, as mentioned earlier.
- Partitioning keywords (P-key): picking words that are used for the partitioning of objects in the W-IBR-tree.
- Non-partitioning keywords (NP-key): picking words that are not used for the partitioning of objects in the W-IBR-tree.

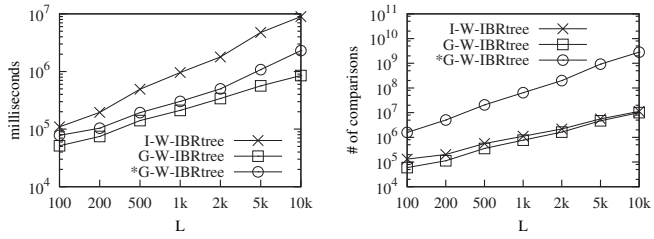
Figure 20 shows the total I/O costs of the methods for these three types of keywords. The W-IBR-tree outperforms the CD-IBR-tree using R-key and P-key, while they have comparable performance for NP-key. The W-IBR-tree beats the Inverted-R-trees and the IR^2-tree for all three types of keywords. R-key uses randomly generated keys and does not favor any particular index. In this setting, the W-IBR-tree achieves better performance than the other indexes.

Effect of Overlap Among Query Keywords.
This experiment evaluates the effect of overlap among keywords in subqueries. A higher overlap means that the subqueries share more common keywords, while a low overlap means subqueries tend to have different keywords.



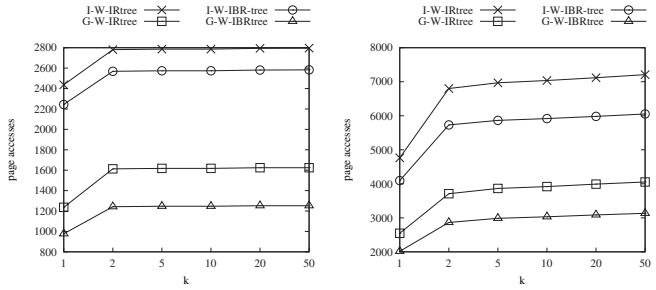
(a) Elapsed Time (b) # of Comparisons

Fig. 17. Computational Optimization on EURO



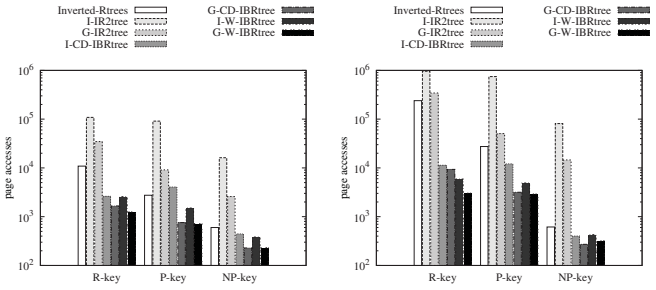
(a) Elapsed Time (b) # of Comparisons

Fig. 18. Computational Optimization on GN



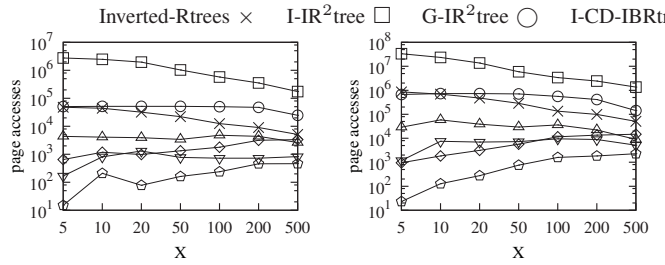
(a) I/O on EURO (b) I/O on GN

Fig. 19. Inverted Bitmap Optimization



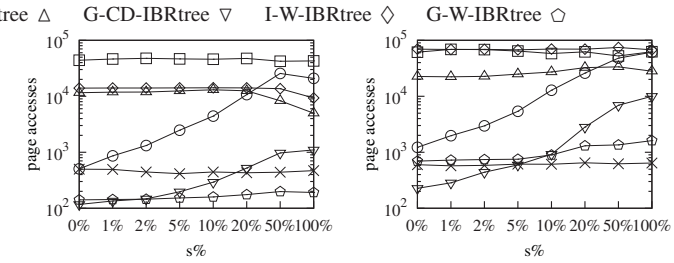
(a) I/O on EURO (b) I/O on GN

Fig. 20. Different Types of Query Keywords



(a) I/O on EURO (b) I/O on GN

Fig. 21. Overlap Among Query Keywords



(a) I/O on EURO (b) I/O on GN

Fig. 22. Side-Length of the Joint Query's MBR

We randomly generate subquery keywords within the top-X most frequent words from the dataset, where $X = 5, 10, 20, 50, 100, 200, 500$. A small X indicates a high overlap, while a large X indicates a low overlap. In order to remove the effect of different locations of subqueries, all queries are assigned the same location.

Figure 21 shows the total I/O cost of the different solutions as a function of X. The I/O cost of the W-IBR-tree increases as X increases. This is because the W-IBR-tree groups objects in terms of words.

When X is small, i.e., the keywords of queries are similar, the I/O cost is low, and vice versa. Observe that the I/O cost of GROUP on the IR^2-tree and the CD-IBR-tree are insensitive to the value of X. This is because the benefit of keyword sharing among subqueries is counteracted by the better punning power of infrequent words at a large X, which means that queries tend to have fewer frequent words. This also explains why the I/O cost of ITERATE-based solutions and the Inverted-R-trees decreases as X increases and the GROUP algorithm drops at $X = 500$. Again, GROUP-W-IBR-tree is the best.

Effect of Side Length of the Joint Query's MBR. This experiment evaluates the effect of the side length of the joint query's MBR. A small side length implies that subqueries are close to each other, while a large side length indicates that the subqueries are far from each other. We draw rectangles taking the whole spatial center location as their center and $s\%$ of the side length of the spatial domain as their side length, where $s\% = 0\%, 1\%, 2\%, 5\%, 10\%, 20\%, 50\%, 100\%$. Each rectangle corresponds to the MBR of a joint query, and the locations of subqueries are generated randomly within the MBR. In order to remove the effect of different keywords of subqueries, all queries are assigned the same one keyword.

Figure 22 shows the I/O costs of the different solutions as a function of $s\%$. We find that the performance of the Inverted-R-trees and the ITERATE-based solutions are insensitive to the size of the joint query's MBR. This is because they process subqueries one by one so that the performance is not affected by the locations of the subqueries. The I/O costs of the GROUP-based methods increases as $s\%$ increases. This is because the GROUP algorithm shares the computation among subqueries so that a smaller size of the joint query's MBR (subqueries are close to each other) favors it.

7 RELATED WORK

We classify existing related indexes into two categories: augmented R-trees and loosely combined R-trees.

Augmented R-Trees.
In the augmented R-tree approach [2], [3], [26], [27], each entry e in a tree node stores a *keyword summary field* that concisely summarizes the keywords in the subtree rooted at e. This enables irrelevant entries to be pruned during query processing. Two augmented indexes were covered earlier [2], [3]. The IR-tree [2] differs from the other augmented trees in two ways. First, the fanout of the tree is independent of the number of words of objects in the dataset. Second, during query processing, only (a few) posting lists relevant to the query keywords need to be fetched. The bR*-tree [26] augments each node with a bitmap and MBRs for keywords. An improvement of this structure, the virtual bR*-tree [27], has also been proposed. Both structures target the mCK query that retrieves m objects of minimum diameter that match given keywords. This query is very different from the top-k spatial keyword query. Furthermore, when the domain of keywords is large, the bitmaps are large and do not fit in the tree nodes of these two trees.

Loosely Combined R-Trees.
Earlier works use loose combinations of an inverted file and an R*-tree [1], [6], [14], [25], [30]. This approach has the disadvantage that it cannot simultaneously prune the search space using both keyword similarity and spatial distance. The Inverted R-tree by Zhou et al. [30] was covered earlier. It is expensive to process a query with multiple keywords, as multiple R*-trees must be traversed. Also, it requires considerable storage space to maintain a separate R*-tree for each keyword. Hariharan et al. [6] present the so-called KR*-tree in which each node is virtually augmented with the set of keywords that appear in the subtree rooted at the node. The KR*-tree-based query processing algorithm first finds the set of nodes that contain the query keywords. The resulting set then serves as the candidate pool for subsequent search. This yields a large (and unnecessary) overhead when the initial pool is large.

8 Conclusions and Future Work

This paper introduces the joint top-k spatial keyword query and presents efficient means of computing the query. Our solution consists of: (i) the W-IBR-tree that exploits keyword partitioning and inverted bitmaps for indexing spatial keyword data, and (ii) the GROUP algorithm that processes multiple queries jointly. In addition, we describe how to adapt the solution to existing index structures for spatial keyword data. Empirical studies with combinations of two algorithms and a range of indexes demonstrate that the GROUP algorithm on the W-IBR-tree is the most efficient combination for processing joint top-k spatial keyword queries.

It is straightforward to extend our solution to process top-k spatial keyword queries in spatial networks. We take advantage of Euclidean distance being a lower bound on network distance. While reporting the top-k objects incrementally, if the current object is farther away from the query in terms of Euclidean distance than is the k^{th} candidate object in terms of network distance, the algorithm stops and the top-k result objects in the spatial network are found. The network distance from each object to a query can be easily computed using an existing, efficient approach [20].

An interesting research direction is to study the processing of joint moving queries, which is useful in environments with continuously moving users.

Acknowledgments

C. S. Jensen is an Adjunct Professor at University of Agder, Norway. Man Lung Yiu was supported by ICRG grants A-PJ79 and G-U807 from the Hong Kong Polytechnic University.

References

[1] Y.-Y. Chen, T. Suel, and A. Markowetz. Efficient query processing in geographic web search engines. In *SIGMOD*, pp. 277–288, 2006.
[2] G. Cong, C. S. Jensen, and D. Wu. Efficient retrieval of the top-k most relevant spatial web objects. In *VLDB*, pp. 337–348, 2009.
[3] I. De Felipe, V. Hristidis, and N. Rishe. Keyword search on spatial databases. In *ICDE*, pp. 656–665, 2008.
[4] M. Duckham and L. Kulik. A formal model of obfuscation and negotiation for location privacy. In *PERVASIVE*, pp. 152–170, 2005.
[5] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *SIGMOD*, pp. 47–57, 1984.
[6] R. Hariharan, B. Hore, C. Li, and S. Mehrotra. Processing spatial-keyword (SK) queries in geographic information retrieval (GIR) systems. In *SSDBM*, p. 16, 2007.
[7] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM TODS*, 24(2):265–318, 1999.
[8] M. Hong, M. Riedewald, C. Koch, J. Gehrke, and A. J. Demers. Rule-based multi-query optimization. In *EDBT*, pp. 120–131, 2009.
[9] H. Hu, J. Xu, W. S. Wong, B. Zheng, D. L. Lee, and W. C Lee. Proactive caching for spatial queries in mobile environments. In *ICDE*, pp. 403–414, 2005.
[10] P. Kalnis and D. Papadias. Multi-query optimization for online analytical processing. *Inf. Syst.*, 28(5):457–473, 2003.
[11] L. Kaufman and P. J. Rousseeuw. *Finding groups in data: an introduction to cluster analysis*. New York: Wiley, 1990.
[12] H. Kido, Y. Yanagisawa, and T. Satoh. An anonymous communication technique using dummies for location-based services. In *IEEE Conf. on Pervasive Services*, pp. 88–97, 2005.
[13] H. Lu, C. S. Jensen, and M. L. Yiu. PAD: privacy-area aware, dummy-based location privacy in mobile Services. In *MobiDE*, pp. 16–23, 2008.
[14] B. Martins, M. J. Silva, and L. Andrade. Indexing and ranking in geo-IR systems. In *GIR*, pp. 31–34, 2005.
[15] M. Murugesan and C. Clifton. Providing privacy through plausibly deniable search. In *SDM*, pp. 768–779, 2009.
[16] S. Naranan and V. K. Balasubrahmanyan. Models for power law relations in linguistics and information science. *Journal of Quantitative Linguistics*, 5(1-2):35–61, 1998.
[17] A. Papadopoulos and Y. Manolopoulos. Multiple range query optimization in spatial databases. In *ADBIS*, pp. 71–82, 1998.
[18] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *SIGMOD*, pp. 249–260, 2000.
[19] F. Saint-Jean, A. Johnson, D. Boneh, and J. Feigenbaum. Private web search. In *WPES*, pp. 84–90, 2007.
[20] H. Samet, J. Sankaranarayanan, and H. Alborzi. Scalable network distance browsing in spatial databases. In *SIGMOD*, pp. 43–54, 2008.
[21] M. Sanderson and J. Kohler. Analyzing geographic queries. In *GIR*, 2 pages, 2004.

[22] T. K. Sellis. Multiple-query optimization. *ACM TODS*, 13(1):23–52, 1988.
[23] Y. Tao, J. Zhang, D. Papadias, and N. Mamoulis. An efficient cost model for optimization of nearest neighbor search in low and medium dimensional spaces. *IEEE TKDE*, 16(10):1169–1184, 2004.
[24] Y. Theodoridis and T. K. Sellis. A model for the prediction of R-tree performance. In *PODS*, pp. 161–171, 1996.
[25] S. Vaid, C. B. Jones, H. Joho, and M. Sanderson. Spatio-textual indexing for geographical search on the web. In *SSTD*, pp. 218–235, 2005.
[26] D. Zhang, Y. M. Chee, A. Mondal, A. K. H. Tung, and M. Kitsuregawa. Keyword search in spatial databases: towards searching by document. In *ICDE*, pp. 688–699, 2009.
[27] D. Zhang, B. C. Ooi, and A. Tung. Locating mapped resources in web 2.0. In *ICDE*, pp. 521–532, 2010.
[28] J. Zhang, N. Mamoulis, D. Papadias, and Y. Tao. All-nearest-neighbors queries in spatial databases. In *SSDBM*, pp. 297–306, 2004.
[29] B. Zheng and D. L. Lee. Semantic Caching in Location-Dependent Query Processing. In *SSTD*, pp. 97–116, 2001.
[30] Y. Zhou, X. Xie, C. Wang, Y. Gong, and W.-Y. Ma. Hybrid index structures for location-based web search. In *CIKM*, pp. 155–162, 2005.
[31] G. K. Zipf. *The Psycho-Biology of Language*. Houghton Mifflin, Boston, 1935.
[32] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2):6, 2006.
[33] S. T. Leutenegger, J. M. Edgington, and M. A. Lopez. STR: A simple and efficient algorithm for R-tree packing. In *ICDE*, pp. 497–506, 1997.

APPENDIX A
ADAPTATION OF ITERATE AND GROUP

We proceed to show that ITERATE and GROUP, with slight adjustments, are also applicable to the existing IR^2-tree that augments the R-tree with signatures [3]. Each leaf entry p stores a signature $p.\alpha$ as a fixed-length bitmap that summarizes the set of keywords in p. Each non-leaf entry e stores a signature $e.\alpha$ that is the bitwise-OR of the signatures of the entries in the child node of e. The IR^2-tree faces the challenge of whether the signatures possess enough pruning power to offset the extra cost incurred by the taller trees that result from the inclusion of the signatures.

Let $q_i.\alpha$ be the signature for the keyword set $q_i.\psi$ of subquery q_i. In the IR^2-tree, each entry stores a signature, which is a bitmap that summarizes the keywords of the objects in its subtree. To determine whether a non-leaf entry e may contain all relevant keywords of q_i, we check their signatures: $q_i.\alpha \subseteq e.\alpha$ (false positives may exist). To determine whether a leaf entry p may contain all relevant keywords of q_i, we first check their signatures: $q_i.\alpha \subseteq p.\alpha$. If the check is positive, we need to retrieve the keyword set of q_i for actual keyword containment checking. Pruning Rules 1 and 3 have to be modified accordingly.

APPENDIX B
LOOSELY COMBINED R-TREES AND INVERTED LISTS

Indexes exist that combine loosely an inverted file and an R*-tree. Zhou et al. [30] evaluate two such combinations and find that the best approach is to build, for each distinct keyword, a separate R*-tree on the objects containing the keyword. We call this index Inverted-R-trees.

Since no algorithm for top-k spatial keyword queries exists for this index, we propose to an algorithm that applies incremental best-first search [7] on multiple trees concurrently. This algorithm employs a priority queue U and a hash table $\mathcal{HT}$.

Priority queue U is used to examine entries in ascending order of their distances to q. Each queue entry is of the form $\langle tree\_id, e, mindist(q, e)\rangle$, where $tree\_id$ is the identifier of a tree and e is an entry in that tree. We initially enqueue the root node of each tree T_i whose associated keyword belongs to the keyword set of q.

Hash table $\mathcal{HT}$ is used to keep track of candidate objects. Each entry is of the form $\langle p, count\rangle$, where p is an object and $count$ is the number of matched keywords in the query. When an object p is dequeued, the corresponding entry in $\mathcal{HT}$ is updated. When $count$ equals the number of keywords in the query, the object is reported as a result. The algorithm stops when k objects are found. The correctness follows because objects are dequeued in ascending distance from q.

Dingming Wu received the bachelor's degree in computer science and the master's degree in computer science from Huazhong University of Science and Technology and Peking University in 2005 and 2008, respectively. She is currently a PhD student at the Department of Computer Science, Aalborg University, Denmark, under the supervision of Prof. Christian S. Jensen. Her research focuses on spatial keyword query processing.

Man Lung Yiu received the bachelor's degree in computer engineering and the PhD degree in computer science from the University of Hong Kong in 2002 and 2006, respectively. Prior to his current post, he worked at Aalborg University for three years starting in the Fall of 2006. He is now an assistant professor in the Department of Computing, Hong Kong Polytechnic University. His research focuses on the management of complex data, in particular query processing topics on spatiotemporal data and multidimensional data.

Gao Cong is an Assistant Professor at Nanyang Technological University, Singapore. He was an Assistant professor at Aalborg University, Denmark from 2008 to 2010. Before that, he worked as a researcher at Microsoft Research Asia, and as a postdoc research fellow at University of Edinburgh. He received his Ph.D. from National University of Singapore. His current research interests include search and mining social media, and spatial keyword query processing.

Christian S. Jensen Ph.D., Dr.Techn., is a Professor of Computer Science at Aarhus University, Denmark, where he leads the Data-Intensive Systems research group. Prior to joining Aarhus, he held faculty positions at Aalborg University for two decades. From September 2008 to August 2009, he was on sabbatical at Google Inc., Mountain View.

His research concerns data management and spans semantics, modeling, indexing, and query and update processing. During the past decade, his focus has been on spatio-temporal data management.

He is a member of the Royal Danish Academy of Sciences and Letters, the Danish Academy of Technical Sciences, and the EDBT Endowment; and he is a trustee emeritus of the VLDB Endowment. He received Ib Henriksen's Research Award for contributions to temporal data management, Telenor's Nordic Research Award for contributions to mobile services and data management, and the Villum Kann Rasmussen Award for contributions to spatio-temporal databases and data-intensive systems.

He is vice president of ACM SIGMOD, an editor-in-chief of the VLDB Journal and has served on the editorial boards of ACM TODS, IEEE TKDE, and the IEEE Data Engineering Bulletin. He was PC chair or co-chair for STDM 1999, SSTD 2001, EDBT 2002, VLDB 2005, MobiDE 2006, MDM 2007, TIME 2008, DMSN 2008, ISA 2010, and ACM GIS 2011. He will PC co-chair for IEEE ICDE 2013.