Efficient Algorithms for Answering Reverse Spatial-Keyword Nearest Neighbor Queries

Ying Lu† Gao Cong‡ Jiaheng Lu♯ Cyrus Shahabi†

†Integrated Media Systems Center, University of Southern California, Los Angeles, CA 90089
‡School of Computer Engineering, Nanyang Technological University, Singapore, 639798
♯School of Information, Renmin University of China, Beijing, 10087
†{ylu720, shahabi}@usc.edu ‡{gaocong}@ntu.edu.sg ♯{jiahenglu}@ruc.edu.cn

ABSTRACT

With the proliferation of local services and GPS-enabled mobile phones, reverse spatial-keyword Nearest Neighbor queries are becoming an important type of query. Given a service object (e.g., shop) q as the query, which has a location and a text description, we return customers such that q is one of top-k spatial-keyword relevant service objects for each result customer. Existing algorithms for answering reverse nearest neighbor queries cannot be used for processing reverse spatial-keyword nearest neighbor queries due to the additional text information. To design efficient algorithms, for the first time we theoretically analyze an ideal case, which minimizes the object/index node accesses, for processing reverse spatial-keyword nearest neighbor queries. Under the derived theoretical guidelines, we design novel search algorithms for efficiently answering the queries. Empirical studies show that the proposed algorithms offer scalability and are orders of magnitude faster than existing methods for reverse spatial-keyword nearest neighbor queries.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications—*Spatial databases and GIS*; H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Performance evaluation*

General Terms

Algorithms, Experimentation, Performance

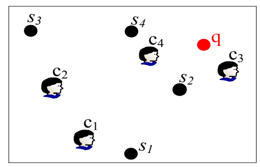
Keywords

Reverse k nearest neighbor, Spatial-keyword query

1. INTRODUCTION

The Internet is acquiring a spatial dimension, with content (e.g., points of interest and Web pages) increasingly being geo-positioned and accessed by mobile users. Therefore, the reverse spatial-keyword nearest neighbor query [5], which considers the fusion of spatial information and textual description, is becoming an important type of queries in the local services of search engines (e.g., Google Maps) and many other websites (e.g., travel planning websites).

SIGSPATIAL'15, November 03 - 06 2015, Bellevue, WA, USA
2015 ACM ISBN 978-1-4503-3967-4/15/11$15.00
DOI: http://dx.doi.org/10.1145/2820783.2820873.



(a) Distribution of customers and shops

Customers	x	y	Specified Keywords
c_1	4	1	(laptop, 1)
c_2	3	4	(camera, 1)
c_3	10	5.5	(laptop, 1)
c_4	7	6	(sportswear, 1)

(b) Preferences of customers in (a)

Shops	x	y	Textual Descriptions
s_1	6	0	(laptop,8), (stationery,7)
s_2	8	4	(laptop,4), (stationery,8)
s_3	2	8	(camera,8), (sportswear,8)
s_4	6	8	(laptop,12), (camera,4)
q	9	7	(laptop,1), (camera,1), (sportswear,8)

(c) Locations and text descriptions of shops in (a)

Figure 1: An example of BRSKkNN queries

Reverse spatial-keyword nearest neighbor queries come in two flavors: Bichromatic Reverse Spatial-Keyword nearest neighbor (BRSKkNN) queries and Monochromatic Reverse Spatial-Keyword nearest neighbor (MRSKkNN) queries. BRSKkNN queries involve two types of objects (e.g., customers and shops), while MRSKkNN queries involve one type of objects (e.g., shops).

Next, we take the BRSKkNN query as an example to explain reverse spatial-keyword nearest neighbor queries. Let $\mathcal{C}$ and $\mathcal{S}$ be the customer set and service set, respectively. Each customer object c in $\mathcal{C}$ has a location $c.p$ and a set of keywords representing the preference of the customer $c.kw$; each service object in $\mathcal{S}$ has a location $s.p$ and a textual description $s.doc$. Given a service object q as the BRSKkNN query, the result will be a set of customers in $\mathcal{C}$ that have q as one of their top-k most *spatial-keyword relevant* objects among the objects in the service set $\mathcal{S}$. Here, the *spatial-keyword relevance* [3,5] is measured by both the spatial proximity to the query location and the text relevance to the query keywords.

$$D_{ST}(c,s) = \alpha(1 - \frac{Dist(c.p,s.p)}{maxD}) + (1-\alpha)\frac{Rel(s.doc|c.kw)}{maxR}, \quad (1)$$

where parameter $\alpha \in [0,1]$ is used to adjust the importance of spatial proximity and the textual relevance at the query time. Normalization constants $maxD$ and $maxR$ denote the maximum spatial distance and textual relevance between customers in $\mathcal{C}$ and shops in $\mathcal{S}$, respectively. $Dist(c.p,s.p)$ is the Euclidian distance between $c.p$ and $s.p$. The text relevance $Rel(s.doc|c.kw)$ of c to s is computed by an information retrieval model. We use the Okapi BM25 model [7], a popular information retrieval model.

Fig. 1 displays the spatial layout and textual descriptions for a set $\mathcal{S}$ of shops and a set $\mathcal{C}$ of customers. Points $c_1,\cdots,c_4$ (in $\mathcal{C}$) shown in Fig. 1(a) represent customers whose locations and keyword preferences are given in Fig. 1(b), and points $s_1,\cdots s_4,q$ (in $\mathcal{S}$) represent shops with locations and texts given in Fig. 1(c) , where the number following a word is the weight, intuitively representing the relevance of a keyword to a shop. Given the shop q as the query object, and $k = 1$, the results of the BRSKkNN query is $\{c_4\}$.

The core problem for efficiently answering BRSKkNN queries is: which objects or index nodes should be visited and in what order to minimize the number of index node / object accesses and thus I/O cost? None of the existing studies can effectively investigate this key problem. The existing solutions (e.g., [2, 4, 8]) developed for RkNN queries without text cannot be used to process RSKkNN queries because they make use of the spatial geometry properties to prune the search space without considering the textual information [5, 6]. The algorithm [5], to our knowledge, which is the state-of-the-art solution for processing MRSKkNN queries (and can be extended to process BRSKkNN queries), prioritizes traversing the top-k spatial-keyword relevant service objects of customers. They assume that visiting the union of these top-k objects will minimize the I/O cost. However, if we consider the problem globally (rather than focusing on a single node E_c), i.e., to find all the result nodes and prune all the non-result nodes, it may suffice to visit a much smaller set of service objects than the set of customer nodes visited by the Algorithm [5]. We illustrate this with the earlier example. The method [5] needs to visit two service objects: s_1 and s_3 (s_1 and s_3 are the most relevant service object of c_1 and c_2, respectively) to prune both customers c_1 and c_2. However, both c_1 and c_2 can be pruned by visiting only s_4, since s_4 is more relevant than q for both c_1 and c_2, though s_4 is not their most relevant service object.

To this end, we analyze an ideal case that aims to minimize the index node accesses for processing the BRSKkNN query. We derive practical guidelines for the following questions, which are crucial for the performance of algorithms for answering the BRSKkNN query: Which customer (resp. service) index nodes or objects must be visited? Which customer (resp. service) index nodes have higher priority to be traversed first so that we can avoid visiting many other nodes? Under the guidelines, we design an efficient solution for answering BRSKkNN queries, which visits the must-be-visited nodes; it prioritizes the visiting order for the other nodes, aiming to reduce node accesses. Results of empirical studies demonstrate the scalability and efficiency of the proposed algorithms: 1) the proposed algorithm for BRSKkNN outperforms the baseline algorithm [5] by 1-2 orders of magnitude, and 2) our algorithm outperforms the algorithm [5] for MRSKkNN by an order of magnitude.

2. BASELINES

Lu et al. [5, 6] proposed a branch-and-bound algorithm for processing the MRSKkNN query, which is referred as our baseline algorithm for the MRSKkNN query, denoted by MoBase. MoBase can be extended to process the BRSKkNN query. We use IR-tree [3] as the spatial-keyword index in the baseline methods and our proposed algorithms for MRSKkNN and BRSKkNN queries. We use two extended IR-trees to organize the two types of objects, customer and service objects. For the baseline method of the BRSKkNN query, denoted by BiBase, we extend the MoBase. BiBase visits the top-k spatial-keyword relevant service objects for each customer entry E_c to determine whether E_c contains the result customers. Specifically, we traverse from the root of the customer index and the root of the service index simultaneously. For each visited customer entry E_c, we estimate the relevance between E_c and all the service entries traversed currently, and use the estimation to update the lower and upper bounds of the spatial-keyword relevance between E_c and its k-th relevant service object. The bounds are used to decide whether E_c can be pruned or contain results: i) E_c is pruned if the maximum relevance between E_c and query object q is smaller than its lower bounds; ii) All the customers in E_c are reported as results if the minimum relevance between E_c and q is larger than its upper bound. Note that BiBase is based on the assumption that top-k spatial-keyword relevant service objects of each customer entry E_c are discriminative in deciding whether E_c contains results or can be pruned. However, if we consider the problem globally as illustrated in Sec. 1, rather than focusing on the top-k most relevant objects for each customer entry, it may suffice to visit a much smaller set of service objects.

3. GUIDE-BASED ALGORITHM

We analyze an ideal case for BRSKkNN queries in Sec. 3.1 and present practical guidelines derived from the ideal case and design efficient guide-based algorithms, denoted by BiGuide, in Sec. 3.2.

3.1 Ideal cases analysis

Many algorithms have been developed for RkNN queries (e.g., [2, 4, 8]). They focus on how to reduce the accesses to index nodes, thus reducing I/O cost and computational cost, which is the key problem for the algorithms of processing RkNN queries. However, none of them tries to analyze the optimal solution such that the accesses to index nodes are minimized.

Next, we define an ideal case for processing BRSKkNN queries, which will identify which entries must be visited, and which entries should be given priority to be visited first to reduce the cost. We assume that customers and service objects are organized in two separate IR-trees. Note that the analysis and the proposed algorithms are applicable if other indexes (e.g., [5]) are in place.

Given a BRSKkNN query q, the minimum set of (customer and service) objects and index nodes that need to be visited to answer q is called the *ideal search region* (*ISR*) of q. A search algorithm that only visits the objects and index nodes in the *ISR* of q is called an ideal case for processing query q. Let $\mathcal{C}$ and $\mathcal{S}$ be the customer and service object databases, respectively. Let C_r be the **result** customer set, i.e., $\forall c \in C_r$, query service object q is the $t+1$-th ($t < k$) nearest object of c. Let C_p be the set of customers that are **not results** of q. Next, we first define the *ISR* for BRSKkNN when objects are not indexed, and then we analyze the case in the presence of indexes.

DEFINITION 1. *Given a service object s and a customer c, we say s* **contributes to** *c (or c is* **contributed by** *s) iff the spatial-keyword relevance between s and c is no less than the relevance between the query object q and c, i.e., $D_{ST}(c,s) \geq D_{ST}(c,q)$.* □

In the absence of an index, all the customers must be visited since each customer $c \in \mathcal{C}$ needs to be accessed to determine whether c is a result. The analysis for service objects is more complicated. Two sets of service objects must be visited:

1) $S_1 = \bigcup_{c \in C_r} TopSK(c, t, \mathcal{S})$: The top-$t$ ($t<k$) service objects for each result in C_r. A customer c can be confirmed to be a result iff fewer than k service objects can contribute to c; thus the top-t nearest service objects of each result c in C_r must be visited to identify c to be a result.

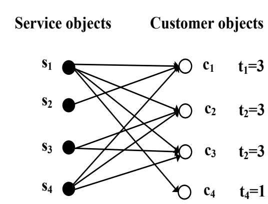


Figure 2: Illustration of the two sets S_1, S_2 for BRSKkNN. An edge from s to c denotes s contributes to c. The number t_i following a customer c_i is the number of service objects that contribute c_i

2) S_2: The minimum contributing set (*MCS*). To prune the non-result customers in C_p, ideally we visit the minimum set of service objects. To prune a customer c we

need to find at least k service objects that can contribute to c. Service objects in S_1 may also contribute to some customers in C_p. We define *MCS* as the minimum set S' of service objects in $\mathcal{S}$ s.t. $\forall c \in C_p$, c can be contributed by at least k service objects in $S' \cup S_1$. Intuitively, *MCS* is the minimum set of service objects that must be visited to prune customers in C_p. The example shown in Fig. 2 illustrates the two parts, S_1, S_2. Suppose k=2, result customer is $C_r = \{c_4\}$, and non-result customers are $C_p = \{c_1, c_2, c_3\}$. We have $S_1 = \{s_1\}$ since s_1 is the top-1 service object for result customer c_4. And we have $S_2 = \{s_4\}$ since $\{s_4\}$ is the minimum set of service objects such that at least k (=2) service objects in $S_1 \cup S_2$ contribute to each non-result customer in C_p.

We proceed to analyze the *ISR* of the BRSKkNN query q in the presence of indexes. We first consider the *ISR* for customer index nodes. We say an index node *contains* customer c if the subtree rooted at the node contains c. If an index node *contains* any result customer (in C_r), we must visit the node since the result customer must be visited. Ideally, we do not visit the index nodes that do not contain result customers. Hence, the *ISR* for customer index consists of all the customer nodes that contain result customers.

For service index, the *ISR* consists of two parts: 1) For each customer c in result C_r, the service index nodes whose maximum spatial-keyword relevance to a customer c in result C_r is equal to or larger than the relevance of query object q to c must be visited to identify c is a result. 2) The minimum contributing set (*MCS*) of service index nodes. Similar to the analysis without index, we want to identify the minimum set of index nodes, denoted by *MCS*, and we can identify the set of non-result customers C_p by visiting nodes in *MCS* and nodes in part 1).

The problem of finding *MCS* is computationally intractable since it can be reduced from the minimum Set-Cover problem, which is an NP-hard problem. Note that the purpose of our analysis is not to develop an algorithm for achieving the ideal case. Nevertheless, the analysis on the ideal case indicates what types of index entries should be visited and the orders of visiting them. These offer practical guidelines for developing efficient algorithms, which have not been explored by the existing studies on BRSKkNN queries.

3.2 Search Algorithm for BRSKkNN

Based on the analysis in Section 3.1, we derive three guidelines: **Guideline 1** $\forall c \in C_r$, we must identify its top-t ($t<k$) most spatial-keyword relevant service objects to identify c as a result customer. Thus, we must visit all the service entries whose maximum spatial-keyword relevance to c is no less than the relevance between c and q. Note that it is non-trivial to estimate the maximum relevance and this will be covered in the full version [1]. **Guideline 2** Customer index nodes that contain result customers must be visited. **Guideline 3** We need to visit service objects that can prune more non-result customers. Ideally, we can visit only the objects in *MCS*. It is desirable to visit such service objects that can contribute more non-result customers to reduce the accesses of service entries.

Under the guidelines, we develop a novel algorithm for BRSKkNN queries. We design different search strategies to process potential result and non-result customers. The algorithm is in two steps: Preliminary Diagnose (PD) Step and Confirmed Diagnose (CD) Step.

In PD Step, we aim to 1) identify customer entries that are likely to contain result customers (which will be further checked and confirmed in CD Step) (guideline 2), 2) identify the service entries that must be visited (guideline 1), and 3) prune non-result customer entries as much as possible by visiting a minimum set of service objects (guideline 3). In CD Step, we aim to 1) find the top-t service objects for result customer entries from the PD step to confirm them to be results (guideline 1), and 2) selectively visit a minimum set of service entries such that they can contribute to the non-result customer entries, thus pruning non-result customers (guideline 3).

3.2.1 Algorithm PD

We first introduce several definitions and lemmas.

DEFINITION 2. *Given a customer entry E_c, its* **contribution number** *is the number of service objects that contribute to each customer in E_c. Further given service entry E_s, we say that service entry E_s* "contributes to" *E_c if $\forall s \in E_s$, $\forall c \in E_c$, s can contribute to c; E_s* "cannot contribute to" *E_c if $\forall s \in E_s$, $\forall c \in E_c$, s cannot contribute to c; otherwise, we say E_s* "may contribute to" *E_c.* □

DEFINITION 3. *Let $\mathcal{T}$ be a set of service entries that do not have ancestor-descendant relationship. For a customer entry E_c, the* ***lower bound*** *of the contribution number of E_c, denoted as LCN_{E_c}, is defined in Eqn(2). And the* ***upper bound*** *of the contribution number of E_c, denoted as UCN_{E_c}, is defined in Eqn(3), where $|E_s|$ is the number of service objects contained in E_s.*

$$LCN_{E_c} = \sum_{\substack{E_s \in \mathcal{T} \\ E_s \text{ contributes to} E_c}} |E_s| \tag{2}$$

$$UCN_{E_c} = \sum_{\substack{E_s \in \mathcal{T} \\ E_s \text{ may contribute to} E_c}} |E_s| \tag{3}$$

LEMMA 1. *Given a customer entry E_c, if $LCN_{E_c} \geq k$, then E_c does not contain any result customer and can be pruned.*

LEMMA 2. *Given a customer entry E_c, if $UCN_{E_c} < k$, then all the customers in E_c belong to results.*

Algorithm 1 PD (*SR*: the root of shop index tree, *CR*: the root of customer index tree, q: query service object)

Output: *result*: the set of BRSKkNN result objects.

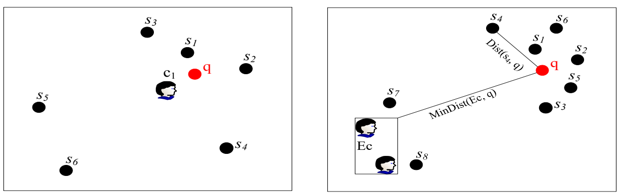
```
1:  U_s ← InitPriorityQueue(SR); L_c ← InitList(CR)
2:  while L_c ≠ ∅ do
3:      (U_s, kNew) ← FindNextkNN(U_s, q)
4:      nkOld ← nkOld ∪ kNew; L_n = ∅
5:      for each entry E_c in L_c do
6:          L_c ← L_c \ E_c
7:          case ← PreferCase(E_c, kNew, q)
8:          if(case = CannotContribute) then L_ca ← L_ca ∪ {E_c}
9:          else if (case = MayContribute) then
10:             if (E_c is an index node) then
11:                 for each child entry CE_c of E_c do
12:                     LCN_{CE_c} = LCN_{E_c}; UCN_{CE_c} = LCN_{E_c}
13:                     L_n ← L_n ∪ {CE_c}
14:             else L_ca ← L_ca ∪ {E_c}
15:     L_c ← L_c ∪ {L_n}
16: result ← CD(EnQueue(U_s, nkOld), L_ca, q)

    Procedure PreferCase(E_c: Customer entry, kNew, q)
17: for each service s in kNew do
18:     if LBΔ((E_c, s) − (E_c, q)) ≥ 0 then
19:         LCN_{E_c} ← LCN_{E_c} + 1; UCN_{E_c} ← UCN_{E_c} + 1;
20:         if LCN_{E_c} = k then Return CanContribute;
21: if ∀s∈kNew, UBΔ((E_c, s) − (E_c, q)) < 0 then Return CannotContribute;
22: Return MayContribute;
```

Next we introduce Algorithm PD (see Algorithm 1). PD works in an iterative manner. In each iteration, it first identify the k most spatial-keyword relevant service objects of query object q to process customer entries to see if a customer entry contains results. Intuitively, service objects that are most spatial-keyword relevant to q are likely to contribute a result customer, and also they are likely to be effective in pruning non-result customers. Then we use the k most relevant service objects to process each customer entry E_c by invoking PreferCase (Line 7). The result of PreferCase will be one of the following three cases:

- `CanContribute`: The k service objects can contribute to E_c. In this case, E_c can be pruned according to Lemma 1 (Line 20).

- `CannotContribute`: All the k service objects cannot contribute to E_c. In this case, we move E_c to CD Step (Line 8). Customer entry E_c satisfying this heuristic follows two possibilities. Consider Fig. 3 as an example, let k=2 and consider spatial information only for intuitive illustration. 1) E_c is likely to be a result entry. As shown in Fig. 3(a), both s_1 and s_2, which are the 2NN of q, cannot contribute to customer c_1 and thus c_1 satisfies the heuristic. We move c_1 to CD Step, which will find q is the nearest neighbor of c_1 and thus identify c_1 as the RkNN result of q. In this way, we can avoid visiting the rest service objects $s_3, \cdots, s_6$. 2) E_c is not a result entry which cannot be pruned effectively by the service objects near query object q. As shown in Fig. 3(b), the 2NN of q: s_1 and s_2 cannot contribute to customer entry E_c, and then we move E_c to CD Step. Therefore, we can avoid visiting service objects $s_1, \cdots, s_6$ which are close to q but cannot be used to prune E_c, while in CD Step, service objects s_7 and s_8 near E_c will be visited to prune E_c.
- `MayContribute`: The k service objects may contribute to E_c. In this case, we visit the child of E_c if E_c is an index node (Line 12); otherwise move E_c to CD Step (Line 14).



(a) E_c (i.e., c_1) contains results (b) E_c does not contain results

Figure 3: Heuristic illustration for customer entry E_c

3.2.2 Algorithm CD

Algorithm CD (see Algorithm 2) progressively computes the lower and upper bounds of contribution number for the customer entries moved from PD by visiting service entries in a branch-and-bound manner to determine whether the customer entries are results. The challenge here is how we can visit as few as possible service entries. To achieve this, CD selectively visits a set of service and customer entries and accesses them in an order based on two priority queues:

1) a max-priority queue U_{s2} which maintains the service entries to be traveled. Each element E_s in U_{s2} is associated with a set $E_s.C$ of customer entries that may be contributed by E_s. The rationale for maintaining $E_s.C$ for E_s is that E_s and its descendant entries are not necessary to be visited for the customer entries that are not in $E_s.C$. In other words, CD only needs to consider the customer entries in $E_s.C$ when processing E_s. The key of an element E_s in U_{s2} is the total contribution number of customer entries in $E_s.C$. This is because service entries contributing to more customer entries are likely to prune more non-result customers (under Guideline 3).

2) CD maintains a max-priority queue U_{ca} on the customer entries that need to be checked whether to be results. Each customer entry E_c is associated with a set $E_c.S$ of service entries that may contribute to E_c. The key of an element E_c in U_{ca} is the total contribution number of service entries in $E_c.S$ to E_c. Intuitively, a customer entry associated with a large number of service entries is likely to be diverse in their spatial and textual information, and thus it is difficult to process the entry as a whole. Hence we prioritize processing such entries (i.e., visiting their component entries). In contrast, the bounds for customer entries associated with fewer service entries are more likely to be tight, and thus it is more likely to determine if such a entry as a whole contains results or not without accessing its component entries.

Algorithm 2 CD (L_s, L_{ca}, q)

1: Initialize two max-priority queues U_{s2} and U_{ca}
2: **for** each customer entry E_c in L_{ca} **do**
3:   **for** each service entry E_s in L_s **do**
4:     UpdateBounds(E_c, E_s, q, U_{s2})
5:   **if** (IsHitOrDrop(E_c) = **false**) then EnQueue(U_{ca}, E_c)
6: **while** $U_{ca} \neq \emptyset$ **do**
7:   $E_s \leftarrow$DeQueue(U_{s2})
8:   **for** each customer entry E_c in $E_s.C$ **do**
9:     $UCN_{E_c} \leftarrow UCN_{E_c} - |E_s|$
10:     **for** each child entry CE_s of E_s **do**
11:       UpdateBounds(E_c, CE_s, q, U_{s2})
12:     **if** (IsHitOrDrop(E_c) = **true**) **then** $U_{ca} \leftarrow U_{ca} \setminus E_c$
13:   $E_c \leftarrow$DeQueue(U_{ca})
14:   **for** each child entry CE_c of E_c **do**
15:     $LCN_{CE_c} = LCN_{E_c}$; $UCN_{CE_c} = LCN_{E_c}$
16:     **for** each service entry E_s in $E_c.S$ in increasing order of $UB\Delta((E_c, E_s) - (E_c, q))$ **do**
17:       UpdateBounds(CE_c, E_s, q, U_{s2})
18:     **if** (IsHitOrDrop(CE_c) = **false**) **then** EnQueue(U_{ca}, CE_c)
19: Return *results*

Procedure UpdateBounds(E_c, E_s, q, U_{s2})
20: **if** $LB\Delta((E_c, E_s) - (E_c, q)) \geq 0$ **then** //E_s *contributes to* E_c
21:   $LCN_{E_c} \leftarrow LCN_{E_c} + |E_s|$; $UCN_{E_c} \leftarrow UCN_{E_c} + |E_s|$
22: **else if** $UB\Delta((E_c, E_s) - (E_c, q)) \geq 0$ **then** //E_s *may contribute to* E_c
23:   $UCN_{E_c} \leftarrow UCN_{E_c} + |E_s|$
24:   Add E_c into $E_s.C$; Add E_s into $E_c.S$
25: **if** ((E_s is an index node) $\wedge$ ($\exists ce \in E_s.C$, $E_s \in ce.S$)) **then**
26:   EnQueue(U_{s2}, E_s)

Procedure IsHitOrDrop(E_c)
27: **if** ($LCN_{CE_c} \geq k$ or $UCN_{CE_c} < k$) **then**
28:   $\forall se \in CE_c.S, se.C = se.C \setminus CE_c$;
29:   **if**($UCN_{CE_c} < k$) **then** $results.add(subtree(E_c))$
30:   return **true**
31: **else** return **false**

---

4. CONCLUSION

In this paper, we study the reverse spatial-keyword nearest neighbor query. We analyze an ideal case, which minimizes the page accesses, for processing BRSKkNN queries. Under the derived guidelines, we design an efficient search algorithm based on a novel search strategy. The search algorithm can also be successfully applied to process MRSKkNN queries. We conducted experimental studies to evaluate the efficiency of our proposed algorithm. The experimental results show that the proposed algorithms significantly outperform the state-of-the-art methods for both BRSKkNN and MRSKkNN queries. Due to space limitation, we included the details of the experiments and the proofs of Lemmas 1 and 2 in our technical report [1].

5. REFERENCES

[1] Efficient Algorithms for Answering Reverse Spatial-Keyword Nearest Neighbor Queries. `http://www.cs.usc.edu/research/technical-reports-list.htm?#2015`.

[2] E. Achtert, H.-P. Kriegel, P. Kröger, M. Renz, and A. Züfle. Reverse k-nearest neighbor search in dynamic and general metric databases. In *EDBT*, pages 886–897, 2009.

[3] G. Cong, C. S.Jensen, and D. Wu. Efficient retrieval of the top-k most relevant spatial web objects. In *PVLDB*, pages 337–348, 2009.

[4] I.Stanoi, M.Riedewald, D.Agrawal, and A.E.Abbadi. Discovery of influence sets in frequently updated databases. In *VLDB*, pages 99–108, 2001.

[5] J. Lu, Y. Lu, and G. Cong. Reverse spatial and textual k nearest neighbor search. In *SIGMOD*, pages 349–360, 2011.

[6] Y. Lu, J. Lu, G. Cong, W. Wu, and C. Shahabi. Efficient algorithms and cost models for reverse spatial-keyword k-nearest neighbor search. *ACM Trans. Database Syst.*, 39(2):13:1–13:46, May 2014.

[7] S. Robertson, S. Walker, S. Jones, M. Hancock-Beaulieu, and M. Gatford. Okapi at trec-3. In *TREC*, pages 109–126, 1994.

[8] Y. Tao, D. Papadias, and X. Lian. Reverse knn search in arbitrary dimensionality. In *VLDB*, pages 744–755, 2004.