An Efficient Query Indexing Mechanism for Filtering Geo-Textual Data

Lisi Chen, Gao Cong, and Xin Cao
School of Computer Engineering, Nanyang Technological University
Singapore
lchen012@e.ntu.edu.sg, gaocong@ntu.edu.sg, xcao1@e.ntu.edu.sg

ABSTRACT

Massive amount of data that are geo-tagged and associated with text information are being generated at an unprecedented scale. Users may want to be notified of interesting geo-textual objects during a period of time. For example, a user may want to be informed when tweets containing term "garage sale" are posted within 5 km of the user's home in the next 72 hours.

In this paper, for the first time we study the problem of matching a stream of incoming Boolean Range Continuous queries over a stream of incoming geo-textual objects in real time. We develop a new system for addressing the problem. In particular, we propose a hybrid index, called IQ-tree, and novel cost models for managing a stream of incoming Boolean Range Continuous queries. We also propose algorithms for matching the queries with incoming geo-textual objects based on the index. Results of empirical studies with implementations of the proposed techniques demonstrate that the paper's proposals offer scalability and are capable of excellent performance.

Categories and Subject Descriptors

H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing—*Indexing methods*

Keywords

geo-textual data, query index, filtering, subscribing

1. INTRODUCTION

Massive amount of data that are geo-tagged and associated with text information are being generated at an unprecedented scale. First, increasing volume of user generated content on the Web is being associated with geo-locations. Example user generated content includes geo-tagged micro-blogs (e.g., Twitter), photos with both tags and geo-locations in social photo sharing websites (e.g., Flickr), and check-in information on places in location-based social networks (e.g., FourSquare). Second, points of interests (POIs), such as shops and restaurant, are increasingly associated with text descriptions in local search services and online yellow pages. According to recent reports, 53% of mobile searches on Bing have local intent, and 20% of Google searches are related to location [2].

These data are featured with both textual content and geo-spatial content, and we refer to them as geo-textual objects. These geo-textual data may come at a high rate (e.g., geo-tagged tweets or geo-tagged photos) or at a slow rate (e.g., new POIs in a local search service). Nevertheless, they can be modeled as continuously arriving streams of geo-textual objects. Users may want to be notified of interesting geo-textual objects during a period of time. For example, a user may want to be informed when tweets containing term "garage sale" are posted within 5 *km* of the user's home in the next 72 hours. The need for such kind of queries, which is already supported by many services, such as Twitter, is obvious. In the example, users are only interested in tweets containing the query keywords and posted within a certain region and a certain time interval, rather than being overwhelmed by a large number of tweets on other events or other areas or other time periods. As another example, a restaurant owner on Yelp may want to be notified when a new user comment containing term "restaurant" or "food" within 2 *km* of the restaurant is posted before Christmas.

We define such queries as Boolean Range Continuous (BRC) Query that consists of three components: a boolean keyword expression, a spatial region, and a time interval. The answer to a BRC query comprises such geo-textual objects that satisfy the boolean keyword expression, fall in the specified spatial regions, and are generated during the specified time interval. With the proliferation of geo-positioning capabilities and the increasing availability of geo-textual objects, there is a need for systems to notify users of interesting geo-textual objects arriving continually according to their demands expressed by BRC queries. A system that implements the functionality must be efficient and scalable. Indeed, it should be able to manage millions of user demands for notification, where a user may issue multiple queries. For example, as of December 2012, there were more than 200 million monthly active Twitter users[1], and Twitter received more than 1.6 billion queries per day. Based on recent statistics, Foursquare has more than 30 million users with 3 billion check-ins[2]. These requirements pose great challenges for building a system to fulfill the need.

To the best of our knowledge, this paper contains the first study of evaluating a stream of BRC queries on a stream of incoming geo-textual objects in real time. In order to efficiently match BRC queries and incoming geo-textual objects, we build a system called BRCQ. In the system, we propose a new hybrid index structure Inverted File Quad-tree (IQ-tree) to manage the stream of BRC

[1]`https://twitter.com/twitter/status/281051652235087872`. Accessed 20 Feb 2013

[2]`https://foursquare.com/about/`. Accessed 20 Feb 2013

queries. The IQ-tree integrates the Quad-tree for organizing the spatial region information of BRC queries and the inverted file for organizing the keyword expression of BRC queries. The IQ-tree is essentially a Quad-tree extended with inverted files. Each node of the IQ-tree is associated with an inverted file that organizes the keyword expression of the BRC queries associated with the node.

A challenging issue here is how to associate the BRC queries with the nodes of the IQ-tree. We develop an algorithm that iteratively decides whether the queries are to be stored onto the current layer of nodes or stored onto the next layer of nodes in the IQ-tree based on the expected I/O cost. We propose a cost model for the purpose, in which we take into account the following two types of cost: (1) matching a stream of geo-textual objects with a stream of BRC queries and (2) updating the index for the incoming BRC queries.

The incoming BRC queries are first stored in a memory buffer, which are organized into a memory-based inverted file. When queries in the buffer need to be written to the IQ-tree, we choose a set of queries based on a cost model and then partition them onto the IQ-tree based on another cost model. When a geo-textual object arrives, we check both the buffer and the IQ-tree to match the object with BRC queries. With the IQ-tree and the cost models, the proposed BRCQ system is able to efficiently match geo-textual objects with BRC queries, and update the IQ-tree index for the incoming BRC queries.

In summary, the paper's contributions are twofold. First, we present the first study on the problem of matching a stream of BRC queries over a stream of incoming geo-textual objects in real time. We develop a new system for approaching the problem. In particular, we propose a hybrid index, called IQ-tree, and novel cost models for managing a stream of BRC queries. We also propose algorithms for matching the BRC queries with incoming geo-textual objects based on the index. Second, we conduct extensive experiments to evaluate the paper's proposals. Results of empirical studies with implementations of the proposed techniques demonstrate that the paper's proposals offer scalability and are capable of excellent performance. Results also show that the proposals significantly outperform two baseline methods developed by us.

The rest of this paper is organized as follows. Section 2 defines the BRC query and the problem. Section 3 presents the BRCQ system, the structure of the IQ-tree index, and the algorithm. Section 4 details the approach to building the IQ-tree and the corresponding cost models. Section 5 reports the experimental study. Section 6 reviews related work and Section 7 concludes the paper.

2. PROBLEM STATEMENT

Definition 1: Geo-Textual Object. A geo-textual object is defined as $o = \langle \psi, l \rangle$, where $o.\psi$ is a set of keywords, and $o.l$ is a spatial point with latitude and longitude. □

In this paper, we consider a stream of geo-textual objects arriving continually. For example, it can be geo-tagged tweets in Twitter, geo-tagged photos with tags in Flickr, geo-tagged blogs, geo-tagged webpages, etc.

Definition 2: Boolean Range Continuous (BRC) Query. A BRC query is defined as $q = \langle \psi, r, t_c, t_e \rangle$, where $q.\psi$ is a set of query keywords connected by AND or OR semantics, $q.r$ represents a spatial region, and $q.t_c$ and $q.t_e$ are the timestamps that indicate the creation and expiration time of q, respectively. We assume that the creation time is the arrival time of the query in our system. □

A user can submit a Boolean Range Continuous (BRC) query to our system, and then he/she continuously receives geo-textual objects that are relevant to his/her query in a timely fashion.

Intuitively, a Boolean Range Continuous (BRC) query is to continually retrieve the geo-textual objects arriving before the expiration time $q.t_e$ such that the retrieved geo-textual objects satisfy the boolean keyword expression $q.\psi$ and are located in the query range $q.r$. Note that the word "Boolean" corresponds to the keyword component, "Range" represents the spatial component, and "Continuous" represents the temporal component. In contrast to general queries that are evaluated once, continuous queries are to be evaluated continuously over a period of time. In this paper, we consider $q.r$ to be a circle. Hence, we represent $q.r$ with the latitude and longitude of the center of $q.r$, and the radius of $q.r$. The techniques proposed in this paper can be easily extended to handle other types of regions.

Example 1: An example BRC query q: Receive all tweets that contain $bicycle$ and $promotion$, and are posted within 10km of the Empire State Building, until April 5, 2013, 18:00. □

In the example query, $q.\psi$ is $bicycle$ AND $promotion$, $q.r$ is a circle of which radius is 10km and center is the location of the Empire State Building, $q.t_e$ is April 5, 2013, 18:00, and $q.t_c$ is the current time. If we ignore the spatial or temporal aspect in the query, tweets containing *bicycle* and *promotion* all over the world will be returned to the user, which will overwhelm the user with many tweets that he/she is not interested in at all. Therefore, all the textual, spatial and temporal information are important to filter a great number of undesirable results.

In the context of matching a stream of incoming BRC queries over a stream of geo-textual objects, there are a large number of queries representing the interests of the users and they must be checked upon the arrival of a new object immediately.

Problem Statement: We tackle the problem of answering a stream of large number of BRC queries, denoted by B, in real time on a stream of geo-textual objects, denoted by O, continuously.

The number of BRC queries in B can be very large. Note that a user may issue multiple BRC queries and each BRC query may last for a period of time. This posts challenges for efficiently matching BRC queries over geo-textual objects. In order to process the matching efficiently, we build the BRCQ system that employs a dynamic index for the BRC queries. The BRCQ system is able to match BRC queries and geo-textual objects in real time, and dynamically update the index for the BRC queries.

3. BRCQ SYSTEM

3.1 Overview

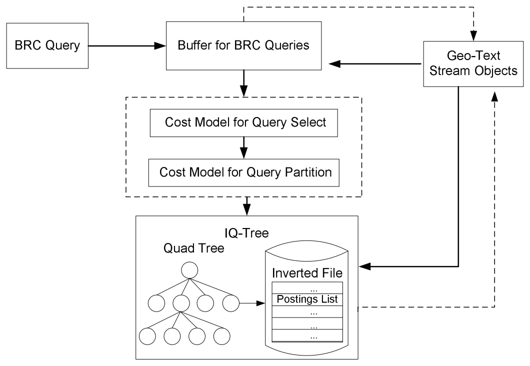


Figure 1: Architecture of the BRCQ System

The BRCQ system is proposed to match a stream of BRC queries with a stream of geo-textual objects. Figure 1 shows the architecture of the proposed system. We propose the IQ-tree for indexing a stream of BRC queries. We propose a cost model to associate the BRC queries and the nodes of the IQ-tree. The cost model concerns with the cost of both (1) matching a stream of geo-textual objects with a stream of BRC queries and (2) updating the index to manage the stream of BRC queries.

It is too expensive to update the IQ-tree whenever a new BRC query comes or a BRC query expires. Instead, we maintain the IQ-tree in a batch way as follows. The incoming BRC queries are first stored in a memory buffer, which are organized into the inverted file. When queries in the buffer need to be written to the IQ-tree, we choose a set of queries based on a cost model and then partition them onto the IQ-tree based on another cost model. When a geo-textual object arrives, we retrieve the corresponding postings lists from both the memory-based inverted file and the disk-based IQ-tree to match the object and BRC queries.

We proceed to describe the structure and the function of individual components in our BRCQ system.

3.2 IQ-tree

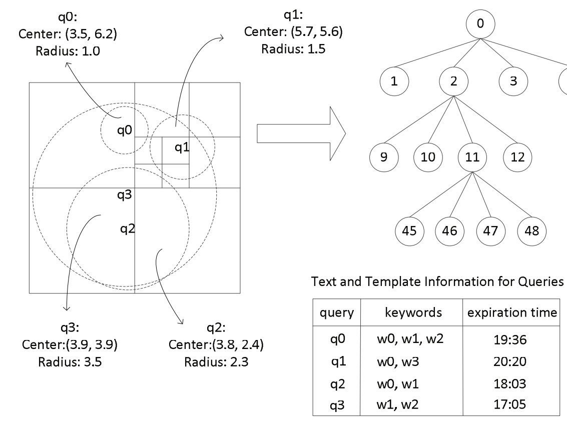
Because both the BRC queries and the geo-textual objects may arrive at a high rate, it is challenging to build an index that can efficiently match a stream of object and maintain the index for the BRC queries dynamically.

Existing geo-textual indices do not consider the temporal information of data and cannot deal with the challenges. They are designed for processing spatial-keyword queries on static geo-textual objects. For example, the IR-tree [8] and SFC-Quad [7] are efficient in processing spatial-keyword queries. However, they are complex and they do not provide an efficient mechanism to maintain the index for the BRC queries with temporal information. The spatial-first grid hybrid index (TS) [19] uses uniform grid cell to index the spatial information of geo-textual objects. TS maintains an inverted file for each cell. It performs worse than the IR-tree [8] and SFC-Quad [7] in processing spatial-keyword queries. However, compared to them, the index structure of TS is simple and is relatively easier to maintain when new data is inserted or deleted from the index. We can extend TS such that it can be used to index BRC queries with the temporal information and the index can be updated dynamically. Nevertheless, it is not efficient in matching stream geo-textual objects to BRC queries. It is used as a baseline solution in our experimental study.

We propose the IQ-tree for indexing a stream of BRC queries. The IQ-tree index is a hybrid geo-textual index. It is essentially a Quad-tree, each node of which is enriched with reference to an inverted file for the BRC queries that are associated at the node.

Quad-tree Component. In the IQ-tree, we use the quad-tree structure to organize the spatial information of the incoming BRC queries. The reasons for using quad-tree are summarized as follows. First, the quad-tree is update-friendly because the query region in a BRC query will be indexed in mutually-exclusive cells. In contrast, the bounding rectangles of the R-tree may overlap with each other, which makes it difficult to use the R-tree to index a stream of BRC queries. Second, we can use different indexing granularity for different BRC queries by considering the spatial distribution and keyword distribution. This is important to the performance of both query update and stream object mapping.

Note that different from many applications that only keep the leaf level quad-cells, we need to keep all the cells including both leaf cells and non-leaf cells. In our implementation, we assign a distinct location code for each Quad-cell in different levels as shown in



Text and Template Information for Queries

query	keywords	expiration time
q0	w0, w1, w2	19:36
q1	w0, w3	20:20
q2	w0, w1	18:03
q3	w1, w2	17:05

Figure 2: Storage Structure of the IQ-tree

Figure 2. Based on location codes, we build a B$^+$-tree to organize the quad-tree cells.

Inverted File Component. Each node of the IQ-tree corresponds to an inverted file for keywords of the BRC queries that are associated at the node. Note that a BRC query may be associated with multiple non-overlapping Quad-cell nodes. The association of BRC queries and the nodes of the IQ-tree is a challenging issue, which affects the system performance significantly. We propose a technique for associating BRC queries with the nodes of the IQ-tree based on cost models, which will be presented in Section 4. An inverted file comprises a vocabulary of terms, and a set of postings lists, each associated with a term.

Definition 3: Posting and Postings List. Each posting records the information of a BRC query that contains term w_i under n_j. A postings list, associated with term w_i under node n_j, contains a set of postings. □

Table 1: Postings List of w_1 under n_{11}

cover type	query id	$q.t_e$	radius of $q.r$	center of $q.r$	$q.\psi \setminus w_1$
PART	0	19:36	1	(3.5, 6.2)	w_0, w_2
PART	2	18:03	2.3	(3.8, 2.4)	w_0
FULL	3	17:05	\	\	w_2

Table 1 shows an example, which illustrates the postings list for term w_1 under node n_{11}. We define two types of structures for a posting, which are treated differently in mapping stream objects to BRC queries. Note that $n.r$ indicates the spatial area represented by the tree node n, and $q.r$ indicates the spatial area of query q.

Definition 4: Fully-Covering. If $n.r \subseteq q.r$, which means that $n.r$ is fully covered by $q.r$, then the query q is a *fully-covering* query w.r.t. tree node n. □

Definition 5: Partially-Covering. If $n.r \cap q.r \neq \emptyset$ and $n.r \backslash q.r \neq \emptyset$, which means that $n.r$ intersects with, but is not fully covered by $q.r$, the query q is a *partially-covering* query w.r.t. tree node n. □

For example, in Figure 2, tree node 46 is fully covered by q_1, and tree node 11 is partially covered by q_1.

cover type	query id	expiration time	additional keywords connected by AND

Figure 3: Posting Structure of Fully-covering Query

If a node is partially covered by a query and we associate the query with the node, we need to record the center and radius of the query region. When an incoming stream object falls in the area

cover type	query id	expiration time	radius of query range	center of query range	additional query keywords connected by AND

Figure 4: Posting Structure of Partially-covering Query

of the node, we use the center and radius information to check whether it falls in the region of the query. However, if a node is partially covered by a query, it is certain that the stream objects fall in the query region if they fall in the area indicated by the tree node. Hence, we do not record the center and radius information.

The two types of posting structure are shown in Figures 3 and 4, respectively. The last field in both types of posting structure is "additional keywords connected by AND." This is to handle the AND semantics for the query keywords in a BRC query. When the keywords of a BRC query have AND semantics, a geo-textual object can be matched to the BRC query only if the object contains all the query keywords. If we use the standard inverted file technique developed for indexing documents, the BRC query will appear in the postings list of each query keyword of the query, and we need to load the postings list of all the query keywords to check if a stream object contains all the query keywords.

To optimize the mapping processing, we apply the *Ranked key method* [23] to organize the query keywords. When the query keywords are connected by AND semantics, we store the query into the postings list of which word is the least frequent among all the query keywords; and the other query keywords are stored in the posting for the query. For example, consider query q_0 for node 1 in Figure 2, query q_0 has three keywords of AND semantics, among which w_2 is the least frequent according to query statistics. Hence, query q_0 is stored in the postings list of w_2, and words w_0 and w_1 are stored in the posting for query q_0 in postings list of word w_2. Note that *Ranked key method* ensures that keywords that frequently appeared take up fewer pages in the inverted file. Consequently, the average object processing I/O cost could be reduced.

If the query keywords of a BRC query are connected by OR semantics, we index the query using standard technique of the inverted file, i.e., the query will appear in the postings list of each keyword in the query.

Each query is associated with its expiration time. When all of the queries in a page expire, the page is marked as an invalid page.

3.3 Buffer for BRC Queries

When a BRC query arrives, we do not immediately write it to the disk-based IQ-tree index. Instead, we store the query into the buffer for BRC queries. We organize the queries in the buffer using the inverted file based on the *Ranked key method* as we discussed in Section 3.2. The postings in a postings list are sorted in ascending order of the expiration time of queries. We set a buffer size M for the total available buffer size, and set a threshold θ for the size of individual postings list. Note that we need to check each BRC query in a postings list to see if it matches with a geo-textual object containing the term corresponding to the postings list. Hence, we set a threshold to avoid very long postings list in memory. When the size of a postings list reaches θ, we choose a set of queries in the postings list according to a cost model, and then partition them into the nodes of the IQ-tree according to another cost model (to be presented in Section 4.2). When the buffer is full, we will process the largest postings list in the buffer. Obviously, inserting individual BRC queries into the IQ-tree will incur much higher I/O cost than inserting them as a batch to the IQ-tree.

3.4 Cost Model

We identify two challenging issues in employing the IQ-tree for indexing the dynamic BRC queries. First, which nodes of the IQ-tree will be associated with a BRC query? In other words, how do we partition a BRC query onto the IQ-tree?

Example 2: The BRC query q_1 in Figure 2 can be partitioned onto various set of nodes in IQ-tree: $\{n_0\}$, $\{n_2\}$, $\{n_9, n_{10}, n_{11}, n_{12}\}$, $\{n_9, n_{10}, n_{12}, n_{45}, n_{46}, n_{47}, n_{48}\}$, etc. Any set of non-overlapping nodes on different levels that intersect $q_1.r$ could be associated with a BRC query. Hence, it is a challenging issue to find a good association with the minimum expected cost. □

Second, which set of queries do we choose from the memory buffer to partition them onto the IQ-tree? We aim to minimize the range of expiration time in one disk page, which will reduce the I/O cost of matching BRC queries with geo-textual objects. Additionally, with a good selection, some queries may expire in buffer, and thus we do not need to write them to disk.

To address the first issue, we propose a cost model (Section 4.1) that takes into consideration the following factors: the query keyword frequency, the query update frequency, and the stream object distribution. To address the second issue, we propose a cost model (Section 4.2) that takes into consideration the following factors: the query temporal information, the query keyword frequency, the query update frequency, and the stream object distribution.

With the cost models, we aim to minimize the expected I/O cost for both matching BRC queries and geo-textual objects and updating the IQ-tree.

3.5 Algorithm for Object Processing

We proceed to present the algorithm for matching a stream of BRC queries indexed by the IQ-tree and a stream of geo-textual objects. Note that we say the tree node n *covers* the object o iff $o.l \in n.r$ and the query q *covers* the object o iff $o.l \in q.r$.

When a stream object o arrives, we need to find the nodes of the IQ-tree whose regions cover the location of object o. Then for each of such nodes, for each keyword w in $o.\psi$, we retrieve its postings list. Each posting in the postings list corresponds to a BRC query q, and we check if the spatial region of query q covers o and the keyword expression of q is matched by o.

Algorithm 1: ObjectProcess(**IQ-tree Index** iq**, Object** o)

```
1  $Result \leftarrow \emptyset$;
2  $n \leftarrow root$;
3  for each keyword $w$ in $o.\psi$ do
4      while $n$ is not a leaf node do
5          for each page $pf$ in postings list of $w$ under node $n$ do
6              if $Max\{q.t_e | q \in pf\} < t_{cur}$ then
7                  Remove $pf$ from postings list of $w$;
8              else
9                  for each query $q$ in $pf$ do
10                     if $q.t_e \geq t_{cur}$ then
11                         if $n$ is fully covered by $q$ then
12                             if $o.\psi$ contains all the other keywords
                               (connected by AND semantics) of $q$
                               then
13                                 $Result$.add($q$);
14                         else
15                             if $o.l$ is in $q.r$ then
16                                 if $o.\psi$ contains all the other
                                   keywords (AND semantics) of $q$
                                   then
17                                     $Result$.add($q$);
18         $n \leftarrow \{n_c | n_c \in n.child$ and $n_c$ covers $o\}$;
19 return $Result$;
```

Algorithm 1 shows the pseudo code for object processing. For each keyword w in o, we start from the root node of the IQ-tree

Table 2: Summary of Notations

Notation	Description
$P(n)$	probability of an object falling in node n
$P(w)$	probability of an object containing keyword w
$n.r$	the spatial area of node n
t_{cur}	current time
$Freq_o$	frequency of incoming objects per time unit
$S_{post}(q,n)$	size of the posting of query q under node n
S_{min}	posting size of fully-covering query with 1 keyword
Q	a batch of BRC queries
$\|Q\|$	the number of a batch of queries to be partitioned
$Q.t_e$	the latest expiration time of queries in Q
$Q.t_e^m$	the earliest expiration time of queries in Q
$S_{frag}(w,n)$	size of the incomplete page of inverted list of w at n
S_{page}	size of a page in disk

and iteratively visit the child node that covers o until we reach the leaf node; at each node, we retrieve the postings list of word w and check if the BRC queries in the list are results (lines 4-18).

After retrieving a page pf (line 5), we check whether the page is expired. If $Max\{q.t_e|q \in pf\} < t_{cur}$, which means that the page pf is expired. Then we remove pf from the inverted list (lines 7-8). If pf is not expired, we proceed to check each query q in the page.

Object o is an answer to query q if the following aspects are satisfied: (1) the expiration time of q is not earlier than t_{cur} (line 10); (2) o is covered by q (lines 11, 15. Note if n is partially covered by q, we need specially check whether $o.l$ falls in $q.r$); (3) o contains all the keywords (connected by AND semantics) in q (lines 12, 16). Note that the *Ranked key method* employed for organizing postings list makes it easy to check this.

We also need to check the BRC queries in buffer, which are organized as in-memory inverted file, to match them with an incoming object o. This can be done by traversing the in-memory postings list of keyword w. For each query stored in the postings list of w, we check whether the three above-mentioned conditions are satisfied, as it is done in lines 9-17 of Algorithm 1.

4. BUILDING IQ-TREE INDEX

We present the techniques for building/updating the IQ-tree. The techniques are used when a batch of queries in a postings list in buffer needs to be written to the IQ-tree. Table 2 summarizes the notations frequently used in the rest of this paper.

4.1 Partitioning Queries onto IQ-tree

We present a new technique that is developed to associate a given batch of BRC queries onto a set of IQ-tree nodes. The objective here is to minimize the expected I/O cost incurred by both matching geo-textual objects with BRC queries and updating the IQ-tree index for the incoming BRC queries.

An incoming BRC query may be fully covered or partially covered by a set of nodes in the IQ-tree, where some nodes are from relatively higher levels in the corresponding IQ-tree, and other nodes are from lower levels. A BRC query can be associated with multiple non-overlapping cells, but not overlapping cells. According to Example 2, the challenging problem here is which nodes should we associate a BRC query.

The tree nodes of the IQ-tree normally can fit into memory easily. Therefore, the I/O cost is incurred by (1) object processing: retrieving inverted lists from disk for matching the BRC queries over incoming geo-textual objects and (2) index update: writing inverted lists to disk for the incoming BRC queries.

We next discuss the I/O cost for object processing. When a geo-textual object arrives, for each keyword contained in the object, we need to retrieve the inverted lists from all the IQ-tree nodes, whose spatial areas cover the location of the object. As shown in Example 2, a BRC query can be partitioned onto both high-level tree nodes and low-level tree nodes. When a query is partially covered by both high-level and low-level nodes, associating it with the high-level nodes will result in higher I/O cost for processing geo-textual objects. The reason is that high-level nodes cover larger spatial regions. Thus, they are more likely to be retrieved when an object comes. In addition, it is more possible that the newly coming object falling in them are not covered by this query as well. Hence, in this case it will waste lots of I/O on reading unrelated queries.

However, when we consider the I/O cost of updating the IQ-tree index, the situation is different. If the BRC queries are partitioned onto many low-level nodes in the IQ-tree, we need to generate a set of postings lists on each node, thus resulting in many postings lists. Therefore, associating queries with low-level nodes will cause higher I/O cost for updating the IQ-tree index. There is a trade-off on associating the BRC queries onto the IQ-tree nodes.

Our system continually receives incoming geo-textual objects and incoming BRC queries. We consider both types of cost in designing the cost model that is used to guide partitioning BRC queries onto the IQ-tree nodes. Here we assume that we are given a batch of BRC queries that are from the in-memory postings list for keyword w. The batch of BRC queries are denoted by Q and we need to update the IQ-tree with the queries in Q. Note that we will discuss how to choose the batch of BRC queries from a postings list in Section 4.2.

Expected I/O for Object Processing. When a batch of BRC queries are inserted into the inverted lists associated with some nodes of the IQ-tree, the expected I/O cost for retrieving inverted lists to process incoming geo-textual objects are increased. This is because the newly inserted queries in the IQ-tree will incur additional I/O.

Consider that we insert a batch of BRC queries, denoted by Q, into the postings list w.r.t. keyword w associated with node n of the IQ-tree. We refer to $C_o(Q,n,w)$ as the expected I/O cost for matching upcoming geo-textual objects with the set of queries Q in the postings list of keyword w under node n. It can be estimated by Equation 1.

$$C_o(Q,n,w) = \lceil \frac{\sum_{q \in Q \wedge q.r \cap n.r \neq \emptyset} S_{post}(q,n) + S_{frag}(w,n)}{S_{page}} \rceil \times (P(n) \times P(w|n)) \times (Freq_o \times (Q.t_e - t_{cur})) \quad (1)$$

In Equation 1, $\lceil \frac{\sum_{q \in Q \wedge q.r \cap n.r \neq \emptyset} S_{post}(q,n) + S_{frag}(w,n)}{S_{page}} \rceil$ represents the number of pages required for storing the batch of BRC queries Q into the inverted list for keyword w under node n. The size of the posting structure of q under n ($S_{post}(q,n)$) is determined according to whether $q.r \subseteq n.r$. Note that not all queries can be stored into a complete page file after partitioning. To reduce the space fragmentation, the incomplete page $S_{frag}(w,n)$ will be combined when we write the new batch of queries into the inverted list. In Equation 1, $P(n) \times P(w|n)$ indicates the probability of an upcoming geo-textual object falling in n and containing keyword w; $Freq_o \times (Q.t_e - t_{cur})$ is the estimated total number of geo-textual objects that will arrive from current time till $Q.t_e$, which is the latest expiration time of queries in Q.

Note that $P(n)$, $P(w|n)$, and $Freq_o$ can be estimated from historical data using maximum likelihood estimation. We assume that these distributions remain stable over a period of time.

Next, we estimate the expected I/O cost for object processing if we store Q into the postings list of keyword w under the four child nodes of n. We can estimate the expected I/O cost for each of the child node, and then sum them up. The cost is denoted by $C'_o(Q,n)$ and is estimated by Equation 2.

$$C'_o(Q,n,w) = \sum_{n.child_i} C_o(Q, n.child_i, w) \quad (2)$$

Expected I/O for Updating IQ-tree When we update the IQ-tree with the incoming BRC queries, it will incur update I/O when we insert the queries to the IQ-tree, and delete them from the IQ-tree when they expire. After we partition a batch of BRC queries Q, which are from the postings list of keyword w, into a set of IQ-tree nodes, postings lists will be generated under each node. We estimate the expected I/O cost for updating the IQ-tree if we store the postings of Q in the postings list of keyword w under node n by Equation 3. Intuitively, it is computed by the number of page accesses. The factor 2 is because we consider the I/O increment for both insertion and deletion (which occurs when the page expires).

$$C_u(Q,n,w) = \lceil \frac{\sum_{q\in Q \wedge q.r\cap n.r \neq \emptyset} S_{post}(q,n) + S_{frag}(w,n)}{S_{page}} \rceil \times 2 \quad (3)$$

If Q are stored into the postings lists of the child nodes of n, the expected I/O for updating the IQ-tree will be the sum of the expected I/O for each of its child node. This is estimated by Equation 4.

$$C'_u(Q,n,w) = \sum_{n.child_i} C_u(Q, n.child_i, w) \quad (4)$$

Total expected I/O cost. If we store the postings of Q in the postings list of keyword w under node n by Equation 3, we compute the total expected I/O cost by summing the two types of expected I/O cost according to Equation 5

$$C_{I/O}(Q,n,w) = C_o(Q,n,w) + C_u(Q,n,w) \quad (5)$$

If the batch of BRC queries Q are stored into the postings lists of the child nodes of n, the expected total I/O cost is computed by Equation 6.

$$C'_{I/O}(Q,n,w) = C'_o(Q,n,w) + C'_u(Q,n,w) \quad (6)$$

Algorithm for associating queries to IQ-tree Next, we present the algorithm for finding a set of IQ-tree nodes to store a batch of queries Q based on the expected I/O cost.

A brute-force approach is to enumerate all the combinations of IQ-tree nodes whose regions overlap with the regions of queries in Q, and then to find the combination with minimum total expected I/O cost based on the proposed cost models. However, it is computationally prohibitive because the number of overlapping nodes can be quite large.

We propose a branch-and-bound algorithm for finding the association with the minimum expected I/O cost. We first introduce two lemmas to be used in our algorithm. The two lemmas together establish the minimum expected cost for partitioning a set of queries Q on the descendant nodes of a node n.

Suppose that the objects falling in node n containing term w are uniformly distributed. As we proceed to partition Q onto the descendant nodes, represented by d, of node n, the minimum possible expected I/O cost for object processing incurred by the objects falling in the spatial area of n can be computed by Lemma 1.

Lemma 1: $minC^d_o(Q,n,w) =$

$$\frac{\sum_{q\in Q} P(q.r\cap n.r) \times P(w|n)}{S_{page}} \times S_{min} \times Freq_o \times (Q.t^m_e - t_{cur}) \quad (7)$$

Proof Sketch: Consider the case of a single query q. Suppose that $N^o_{q,n}$ is the optimal set of nodes that split $q.r \cap n.r$ under n , we use $C^d_o(q,n,w)$ to denote the minimum estimated I/O cost for object processing w.r.t. q (where d indicates the descendant nodes of n), and it can be computed by the following equation:

$$C^d_o(q,n,w) = \frac{\sum_{n_d \in N^o_{q,n}} P(n_d) \times P(w|n_d) \times S_{post}(q,n_d)}{S_{page}} \times Freq_o \times (q.t_e - t_{cur})$$

It is obvious that the area of $q.r$ is smaller or equal to that of $N^o_{q,n}$. Since the objects containing w are uniformly distributed in n, the probability of an incoming object containing w falling in a particular area within n is proportional to the size of the area. Hence, we have:

$$P(n.r\cap q.r) \times P(w|(n.r\cap q.r)) \leq \sum_{n_d\in N^o_{q,n}} (P(n_d) \times P(w|n_d))$$

On account of the uniform distribution of objects containing w in n, we have $P(w|(n.r \cap q.r)) = P(w|n)$ and $P(w|n_d) = P(w|n)$. And $\forall q, n$, we have $S_{min} \leq S_{post}(q,n)$ (fully-covering queries with single keyword have the smallest posting structure). Thus, we can conclude the following equation:

$$C^d_o(q,n,w) \geq \frac{P(n.r\cap q.r) \times P(w|n)}{S_{page}} \times S_{min} \times Freq_o \times (q.t_e - t_{cur}) \quad (8)$$

If $q \in Q$, then $Q.t^m_e - t_{cur} \leq q.t_e - t_{cur}$. Thus, based on Equation 8, we can derive Equation 7 for the multiple queries Q. □

Let $Q.r = \bigcup_{q\in Q} q.r$. If we partition Q onto the descendant nodes of node n, the minimum I/O cost for storing Q denoted by $minC^d_u(Q,n,w)$ can be estimated by Lemma 2.

Lemma 2: $minC^d_u(Q,n,w) =$

$$max\{\lceil \frac{\sum_{q\in Q} S_{post}(q,n)}{S_{page}} \rceil, |\{n_c \in n.child | n_c \cap Q.r \neq \emptyset\}|\} \quad (9)$$

Proof Sketch: Suppose that $N^u_{Q,n}$ is the optimal set of nodes that split $q.r \cap n.r$ under n with minimum expected I/O for storing Q, and the I/O cost can be expressed by the following equation:

$$C^d_u(Q,n,w) = \sum_{n_d\in N^u_{Q,n}} \lceil \frac{\sum_{q\in Q\wedge q.r\cap n_d.r\neq\emptyset} S_{post}(q,n_d)}{S_{page}} \rceil$$

As each query $q \in Q$ will be stored in one or more descendant nodes of n, we have: $C^d_u(Q,n,w) \geq \lceil \frac{\sum_{q\in Q} S_{post}(q,n)}{S_{page}} \rceil$.

On the other hand, it is obvious that the I/O cost for storing Q onto the descendant nodes of n is no less than the number of child nodes of n that intersect with $Q.r$. The reason is that if $Q.r$ intersects with n_c, where $n_c \in n.child$, at least one I/O will be incurred to store Q onto the node n_c or the descendant nodes of n_c. Hence, we have: $C^d_u(Q,n,w) \geq |\{n_c \in n.child | n_c \cap Q.r \neq \emptyset\}|$. Finally, we can conclude Equation 9. □

Based on the two lemmas, we are ready to derive the minimum expected I/O cost for partitioning a set of queries Q on the descendant nodes of a node n.

Lemma 3: If the geo-textual objects containing term w are uniformly distributed, the minimum expected I/O increment incurred by partitioning a set of queries Q on the descendant nodes of a node n is defined by Equation 10.

$$C^u_{min}(Q,n,w) = C^d_u(Q,n,w) \times 2 + C^d_o(Q,n,w) \quad (10)$$

If the geo-textual objects containing term w are non-uniformly distributed on node n, the minimum expected I/O increment is defined by Equation 11.

$$C^{nu}_{min}(Q, n, w) = C^{d}_{u}(Q, n, w) \times 2 \tag{11}$$

Proof Sketch: The uniform case can be derived directly from Lemma 1 and 2. However, for the non-unform case, the minimum cost for object processing is 0, and thus we have Equation 11. □

We need a method of estimating whether the geo-textual objects follow uniform distribution within region of a node n. We can only make the estimation based on the historical data. The problem is to determine whether the historical objects containing keyword w and falling in n are distributed uniformly or not. Considering the fact that the objects distribution is more convincing when the statistics are based on a large number of objects, we set a threshold τ that indicates the probability of an arbitrary upcoming objects containing w and falling on node n. If $P(n) \times P(w|n) < \tau$, we assume that objects on n are uniformly distributed. Otherwise, we consider n to be a node where objects are distributed differently over its four child nodes. Therefore, a more relaxed pruning condition, $C_{I/O}(Q, n, w) \leq C^{nu}_{min}(Q, n, w)$, will be applied when n is a relatively "elusive" area with fewer objects "visited".

Now we are ready to present the algorithm for query partition. We aim to find an optimal set of IQ-tree nodes to associate the batch of queries. Starting from the root node n, we compute the expected I/O cost if queries in Q are stored under node n. Then we recursively partition queries on n's child nodes and compute the expected I/O cost if queries are stored under each child node of n. Based on Lemma 3, we stop partitioning on n's child nodes if the minimum expected I/O cost for storing Q under the n's descendants is larger than that for storing Q under n.

Algorithm 2: FindOptimalNodeset(Q, n, τ)

```
1  $S \leftarrow \emptyset$;
2  if $P(n) \times P(k|n) < \tau$ then $C_{min} \leftarrow C^{u}_{min}(Q, n, w)$;
3  else $C_{min} \leftarrow C^{nu}_{min}(Q, n, w)$;
4  if $C_{I/O}(Q, n, w) > C_{min}$ then
5      for each $n_c \in n.child \wedge n_c.r \cup Q.r \neq \emptyset$ do
6          $S_c \leftarrow$ FindOptimalNodeset($Q, n_c, \tau$);
7          $S \leftarrow S \cup S_c$;
8      if $C_{I/O}(Q, n, w) \leq \sum_{n_i \in S} C_{I/O}(Q, n_i, w)$ then
9          $S \leftarrow \emptyset$;
10         $S.add(n)$;
11     return $S$;
```

The pseudo code for partitioning queries Q onto node n's child node is presented in Algorithm 2. The algorithm takes a batch of queries Q, an IQ-tree root n, and a parameter τ as input. The algorithm first initializes the result node set S, and n is initialized as the root node of $tree$ (line 1). Next, we check whether the objects falling in n with keyword w should be considered to be uniformly distributed in terms of the four child nodes of n. If $P(n) \times P(w|n) < \tau$, we consider it to be uniformly distributed and set the minimum expected I/O C_{min} as $C^{u}_{min}(Q, n, w)$ (Equation 11). Otherwise we set the minimum expected I/O as $C^{nu}_{min}(Q, n, w)$ (Equation 10) (lines 2–3).

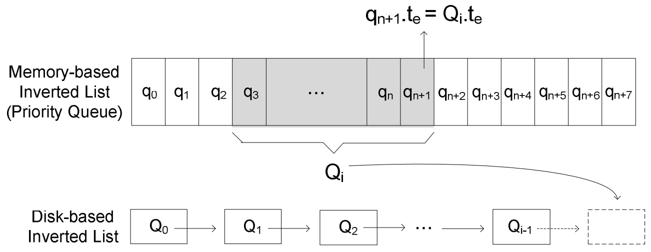
Then we check if $C_{I/O}(Q, n, w)$ (Equation 5) is greater than the minimum expected I/O cost C_{min} after splitting n (line 4). If so, we proceed to compute the optimal node set if Q is partitioned. For each child node n_c of n, we recursively invoke Algorithm FindOptimalNodeset(Q, n_c, τ) to find the optimal set of nodes under n_c. Then the results under each n_c are combined and the optimal set is S (lines 5–7).

To check whether a better partition can be found if we recursively partition Q under n, we compare $C_{I/O}(Q, n, w)$ to the expected I/O of the optimal partition achieved by recursively splitting n, which is stored in S. If $C_{I/O}(Q, n, w)$ is better, we replace the current optimal partition with n (lines 8–10).

4.2 Selecting Query Batch

We present the approach to selecting a batch of queries of a specified size from an in-memory postings list. The batch of queries will be passed to the Algorithm presented in Section 4.1, which is employed to update the IQ-tree with the batch of selected queries.

Recall from Section 3.3 that the query batch selection is triggered as the size of an inverted list in the buffer reaches a specified threshold θ, or the total size of all inverted lists reaches a specified threshold M (the largest inverted list will be selected). Let w denote the term corresponding to the selected inverted list. As introduced in Section 3.2, the postings (each corresponding to a BRC query) in a postings list in buffer are sorted in ascending order of the query expiration time.



$q_{n+1}.t_e = Q_i.t_e$
Memory-based Inverted List (Priority Queue)
q_0 q_1 q_2 q_3 ··· q_n q_{n+1} q_{n+2} q_{n+3} q_{n+4} q_{n+5} q_{n+6} q_{n+7}
Q_i
Disk-based Inverted List
Q_0 Q_1 Q_2 ··· Q_{i-1}

Figure 5: Selecting a batch of queries Q

The problem of query batch selection is to choose a set of queries of size $|Q|$ in the postings list of w in the buffer. This can be illustrated by Figure 5. If the number of queries in the selected postings list is smaller than $|Q|$, all the queries in the list are selected.

Choosing the set of queries from the beginning for which the total size of postings is $|Q|$ will have the minimum expected I/O cost for matching objects with BRC queries according to Equation 1. This is because such a set of queries will have the earliest expiration time among all the candidate batches of size $|Q|$. However, for the queries that will expire soon, we should not write them to disk since they may expire in memory without incurring any I/O cost.

In addition, after a set of queries in a postings list in buffer are selected to write to the IQ-tree on disk, the inverted list may become full again. And then the next round of selecting a batch of queries and writing them to the IQ-tree will occur again. If the time interval between two consecutive rounds of processing is small, it causes more I/O in unit time. Consequently, we should minimize the frequency for updating the IQ-tree with the BRC queries from the buffer.

We aim to minimize the expected I/O cost over a period of time T by considering the cost incurred by the future pages that are selected and written to disk. Thus, we estimate the expected I/O cost in a unit time. We assume that over a period of time T the distributions of queries in terms of spatial part, textual part, and expiration time remain stable.

Now, the problem is to select a batch of queries of size $|Q|$. We denote by $Q.t_e$ the expiration time of the last query in Q, and $Q.c$ the time of selecting Q. Let $Intv(Q, w)$ represent the time interval between two consecutive selected batches from the inverted list for keyword w. We next present how to estimate $Intv(Q, w)$. First, we describe some notations. Let $NumExp(w, Q)$ be the number of queries in the inverted list for w excluding the queries in Q that will expire before the next round of batch query selection. We estimate the time of the next round of batch query selection by the time of the last batch query selection. Let $QRate(w)$ represent the rate of query arrival in the inverted list for keyword w, and it can be estimated by the ratio of the number of queries arriving between the last update and the current time to the length of the interval. We have Equation 12:

$$Intv(Q, w) = \frac{|Q| + NumExp(w, Q)}{QRate(w)}, \tag{12}$$

where $|Q|$ is the number of queries in the selected batch Q, $|Q| + NumExp(w, Q)$ is the expected number of queries generated between two batch selections, i.e., $Intv(Q, w)$.

Let $\Delta C_o(w)$ denote the average I/O cost for object processing per batch of queries over a period T for keyword w. Let $\Delta C_u(w)$ denote the the average I/O cost for updating index per batch of queries over a period T. Then we compute the expected I/O cost in a unit time by Equation 13.

$$D(Q, w, t_e) = \frac{Q.t_e - Q.c}{Intv(Q, w)} \times \Delta C_o(w) + \frac{\Delta C_u(w)}{Intv(Q, w)}, \tag{13}$$

where $(Q.t_e - Q.c)/Intv(Q, w)$ indicates the expected number of selected batches for inverted list of keyword w during the period $Q.t_e - Q.c$.

We can choose $Q.t_e$ to minimize $D(Q, w, t_e)$ based on Equation 13. Note that $\Delta C_o(w)$ and $\Delta C_u(w)$ do not affect the selection since they are the same for any t_e.

We next introduce Lemma 4. According to the lemma, the optimal batch of queries that exhibit the minimum object processing I/O cost in unit time is a consecutive range of queries in inverted list sorted by $q.t_e$.

Lemma 4: Let PQ be a list of queries $q_0, q_1, q_2, ..., q_n$ sorted in ascending order of their expiration time, and $Q \subseteq PQ$. Suppose that $Q' = \{q_i, q_{i+1}, ..., q_{i+j-1}, q_{i+j+1}, q_{i+j+2}, ..., q_k\}$ $(0 \leq i, k \leq n)$ and $Q = \{q_{i+1}, q_{i+2}, ..., q_{i+j-1}, q_{i+j}, q_{i+j+1}, ..., q_k\}$, where Q and Q' have the same size, and the queries in Q' are not contiguous, but the queries in Q are, the expected cost of Q' is higher than that of Q.

Proof Sketch: It is easy to see that the number of queries that will be expired before the next operation of query partition in Q is not less than that in Q', which means that $NumExp(w, Q) \geq NumExp(w, Q')$. According to Equation 12, we have $Intv(Q, w) \geq Intv(Q', w)$. Thus, we have $D(Q, w, q_k.t_e) \leq D(Q', w, q_k.t_e)$ according to Equation 13 □

Based on Lemma 4, we only need to consider consecutive range of queries when choosing $Q.t_e$ to minimize $D(Q, w, t_e)$ according to Equation 13. The idea is implemented in Algorithm 3.

Algorithm 3: BatchSelection($|Q|$, w, pl)

```
1 $optCost \leftarrow +\infty$;
2 $optSet \leftarrow \emptyset$;
3 for each batch of consecutive queries of size $|Q|$ in pl do
4     $t_e \leftarrow$ the expiration time of the last query in the batch if
      $D(Q, w, t_e) < optCost$ then
5         $optCost \leftarrow D(Q, w, t_e)$;
6         $optSet \leftarrow Q$;
7 Delete the queries in $optSet$ from pl;
8 return $optSet$;
```

The algorithm takes three arguments: the size of batch $|Q|$, the selected keyword w, and the corresponding postings list of w. We initialize $optSet$ and $optCost$ (lines 1–2), which stand for the batch of selected queries and the expected unit I/O cost of the selected batch, respectively. We scan the inverted list pl in a pass; for each batch of consecutive queries we compute $D(Q, w, t_e)$ based on Equation 13 (lines 3–6). Finally, we delete the queries in $optSet$ from inverted list pl of word w in buffer, and the batch of queries $optSet$ with the lowest expected cost is returned (lines 7–8).

5. EXPERIMENTAL STUDY

5.1 Baseline

We discuss how to exploit existing techniques to index a stream of BRC queries and process incoming geo-textual objects. No baseline structure and algorithm exist for indexing a stream of BRC queries. Hence we modify and extend some existing geo-textual index structures for the purpose.

Grid plus Inverted File (GIF): A straightforward baseline is to adapt Spatial Primary Index (ST) [19]. Queries stored in disk are partitioned by a global grid index where cells are of the same size. Then for each grid cell, an inverted file is created for indexing the text components of the queries falling in the cell.

Likewise, the GIF index can also work by pruning by text first and then location. Such text-first GIF indexing structure contains separate grid indices for every keyword.

Memory based inverted file is also maintained in buffer. When a query arrives, we first move it into the buffer. For buffer management and batch query selection, we use the proposed technique in this paper for the baseline method.

The major issue of GIF is how to choose the grid resolution. In our experiment, we use two GIFs with different grid resolutions that exhibit diverging performance, namely 20×20 and 100×100.

We use Grid-20 and Grid-100 to denote the spatial-first GIF indices with resolution of 20×20 and 100×100 respectively. And we use Grid-20t and Grid-100t to denote the text-first GIF indices.

Quad-tree plus Inverted File (QIF): This baseline uses a global quad-tree to index the regions of queries. Then, an inverted file is created for each tree node. For buffer management and batch query selection, we use the same techniques as the other methods.

The open question is how to partition a query onto the nodes of the Quad-tree, given a batch of selected queries Q. A heuristic method is that for each $q \in Q$, we compute the overlap area between the query region and a tree node, and then compare the size of the area (denoted by $q.r \cap n.r$) with the size of the tree node. If $\frac{q.r \cap n.r}{n.r} < \Phi$, where Φ is a pre-defined threshold, then we compute $q.r \cap n_c.r$ for each $n_c \in n.child$. The heuristic partition algorithm starts from the root node and runs recursively. It stops when all of the nodes n containing q satisfy $\frac{q.r \cap n.r}{n.r} \geq \Phi$. After finding the optimal set of nodes for each q in Q, we write these queries into the inverted lists at the the corresponding nodes at one time. We try different values for Φ and find that 0.5 yields the best performance in our experiment (Figure 9).

5.2 Experimental Settings

Datasets and BRC queries. Our experiments are conducted on two datasets: 4SQ and TWEETS.

4SQ is a real-life dataset collected from Foursquare, which contains 1,398,238 users and 4,000,000 worldwide check-ins with both location and text information. For each check-in of a user, we generate a query, where the query keywords are the check-in text, and the query range is the minimum circle that covers all the check-ins of the user. If a user has only one check-in, we set the query range as a circle with its center being the location of the check-in, and the radius of query range is randomly assigned between 10km to 100km. Note that the check-in text of the user indicates the interest of the user and region represents the active region of the user, and thus the query generated in this way would be close to real queries. In addition, each check-in text together with its geo-location is regarded as a geo-tagged object.

The dataset TWEETS comprises 20 million tweets collected from users in the U.S.A. Since only 1 percent of the tweets are geo-tagged, we randomly select locations in a real spatial dataset that models the road networks in the U.S.A to associate with tweets without geo-locations. The center of the query range is assigned by a randomly selected node in the dataset of US road networks. The radius of query range is also randomly assigned between 10km to 100km. For each query, we randomly pick an object from the set of tweets and choose a specified number of words as the query keywords. For experiments that are not focusing on the number or the connection type of query keywords, the queries are generated with random number of keywords ranging from 1 to 4, and the keywords in each query is connected by either AND or OR semantics.

For both datasets, query expiration time is assigned according to the Chi-squared distribution, which is defined by $(q.t_e - q.t_c) \sim \chi^2(k)$, where k indicates the average query duration. In our experiments, k is set to 1000.

We set the default value of θ to 32KB and $|Q|$ to 1000, which exhibit the best performance based on Figure 19. And we set the buffer size M at 32MB and 128MB for 4SQ and TWEETS, respectively. We use the LRU cache only for the set of experiments that study the effect of cache size, but not for other experiments.

Setup. For each index structure, we evaluate the performance on object processing and index update. Three metrics are used: (i) runtime, (ii) I/Os, and (iii) index size on disk. We vary the following aspects to evaluate the performance.

A1 Time effect. Each index runs for 2,000 and 20,000 seconds on 4SQ and TWEETS respectively. During each second, 2,000 and 5,000 queries are issued for 4SQ and TWEETS respectively. We report the runtime and I/O cost for processing an object and updating 1,000 queries during the period 1,000 seconds, respectively. We also report the index size on disk.

A2 Effect of partitioning threshold Φ. For QIF index, we vary the value of Φ from 0.1 to 0.7.

A3 The number of query keywords. We separately use AND and OR semantics to connect each query keywords. The number of query keywords varies from 1 to 4.

A4 The number of object keywords. We vary the number of terms in an object from 5 to 20.

A5 Query range size. The radius of query range varies from 10km to 100km.

A6 Buffer size. For each index, we vary the total size of available buffer from 16MB to 256MB.

A7 The number of indexed queries. To evaluate the scalability for each index, we vary the number of queries that are indexed by each index structure, including queries on disk and in buffer.

A8 Effect of LRU cache size. we use an LRU buffer, and vary the cache size from 16MB to 128MB.

A9 Value of $|Q|$. The parameter $|Q|$ indicates the number of queries to be processed in a batch and we vary it from 500 to 2000.

A10 Effect of θ. The size of individual postings list in buffer is varied from 16KB to 256KB.

5.3 Experimental Result

A1 Time Effect: This round of experiments is to evaluate the space, runtime, and I/O cost for object processing and index update when we maintain each index structure under the stream of incoming queries and objects.

We notice that both IQ-tree and Grid-100 perform well and they bear similar performance in object processing as shown in Figure 6. However in term of index update and space cost, as illustrated in Figures 7 and 8, Grid-100 performs much worse than the other indices while IQ-tree and Grid-20 exhibit good performances. The reasons could be explained as follows.

For the grid index, as an object arrives we directly pinpoint to a relatively small grid cell where the object is located. Since the queries intersecting a small cell are less than the queries intersecting a large cell on average, the grid index with finer granularity will exhibit a lower cost for processing objects. However, the good performance is at the expense of additional cost in disk space and index update. While a batch of queries are written to disk, the grid index with finer granularity is required to store queries into more cells. On the contrary, the grid index with coarser granularity performs better in index update, but worse in object processing.

Compared to QIF, IQ-tree performs consistently better in terms of both object processing and index update. Note that both QIF and IQ-tree adopt Quad-tree to organize spatial information. However, QIF is based on a heuristic method to associate BRC queries to the index nodes. In contrast, IQ-tree employs cost model to associate queries with the set of tree nodes in different layers with minimum expected I/O cost for object processing and index update. Consequently, IQ-tree is able to exhibit a good performance in terms of both object processing and index update.

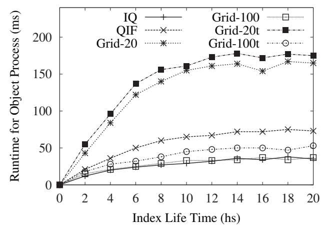
We also observe that the two text-first GIF based indices, namely Grid-20t and Grid-100t, perform consistently worse than Grid-20 and Grid-100, respectively. The reason is that we need to construct a separate grid index for each keyword. Then, the total number of cells is very large. Consequently, it will incur extra I/O costs that may degrade the performance when the grid indices are stored in disk. Since the space costs for text-first GIF indices are very high, especially for Grid-100t, which is 10 times higher than those of the other methods, we do not report their index sizes in Figure 8 for making the figures presentable. We ignore Grid-20t and Grid-100t in the rest of experiments since they perform worse than Grid-20 and Grid-100t, respectively.

The rest of the experiments are only reported on dataset TWEETS due to the space limitation.

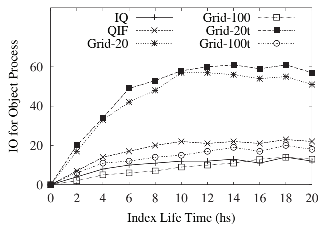
A2 Effect of Φ: In this experiment, we investigate the effect of partitioning threshold Φ for the QIF index. From Figure 9, we observe that as the value of Φ increases the performance of object processing becomes better while the performance of index update becomes worse. If we increase the value of Φ , queries are inclined to be stored into a large number of nodes in the lower layers of Quad-tree. Therefore, much more I/Os are required to store queries onto the Quad-tree. However, since the spatial information of queries is more precisely recorded in the nodes in the lower layers of Quad-tree, the cost for object processing can be decreased.

A3 Effect of the Number of Query Keywords: We proceed to evaluate the index update performance as we vary the number of query keywords connected by OR and AND semantics, respectively. The results for runtime and I/O cost are consistent, and we only show the results for I/O cost due to the space limitation.

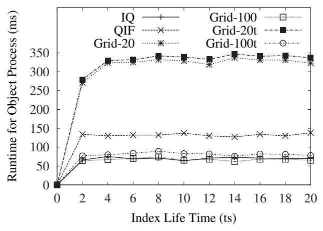
Figure 10 shows that if the query keywords are connected by OR semantics, all of the indices present an increasing trend in index update as we vary the number of keywords. Nevertheless, if the keywords are connected by AND, the performance of each index presents a constant or even slight decreasing trend (Figure 11). This could be explained by the posting structure presented in Section 3.2. Namely, a query of which keywords are connected by OR



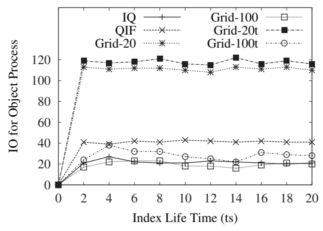
(a) Runtime on 4SQ



(b) I/O on 4SQ

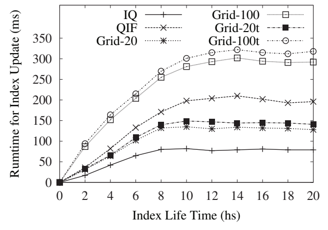


(c) Runtime on TWEETS

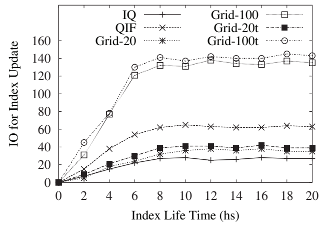


(d) I/O on TWEETS

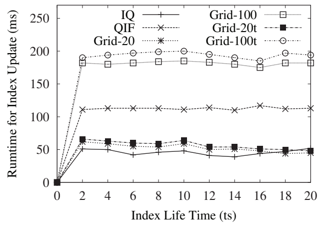
Figure 6: Effect of Time for Object Process



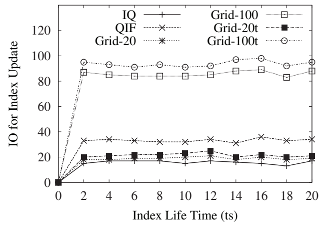
(a) Runtime on 4SQ



(b) I/O on 4SQ

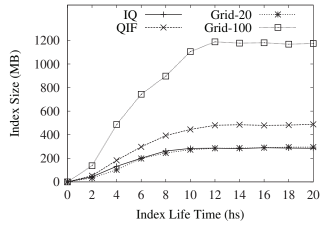


(c) Runtime on TWEETS

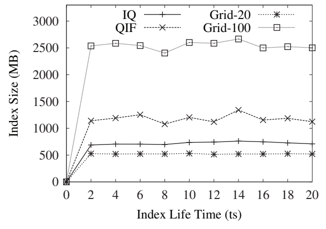


(d) I/O on TWEETS

Figure 7: Effect of Time for Index Update

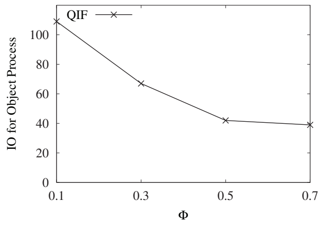


(a) Index Size on 4SQ

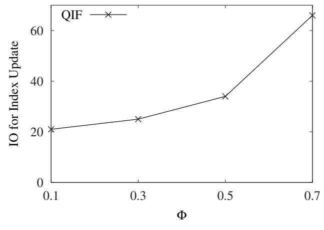


(b) Index Size on TWEETS

Figure 8: Effect of Time for Index Size

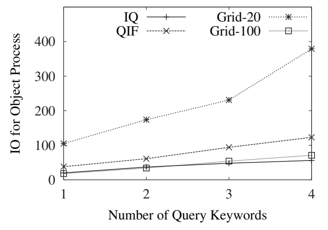


(a) I/O for Object Process

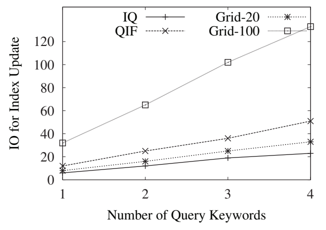


(b) I/O for Index Update

Figure 9: Effect of Φ

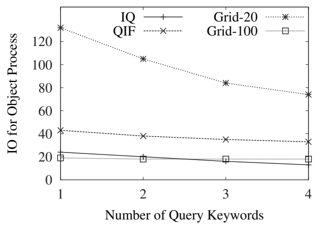


(a) I/O for Object Process

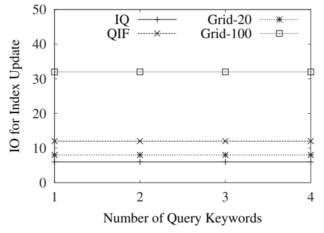


(b) I/O for Index Update

Figure 10: Query Keywords connected by OR

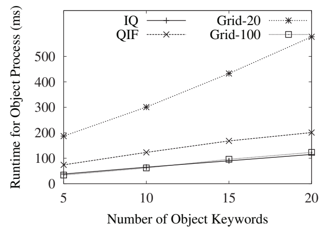


(a) I/O for Object Process

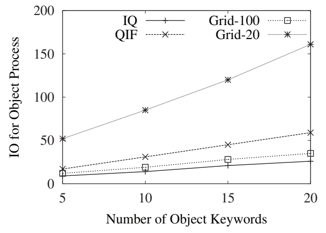


(b) I/O for Index Update

Figure 11: Query Keywords connected by AND

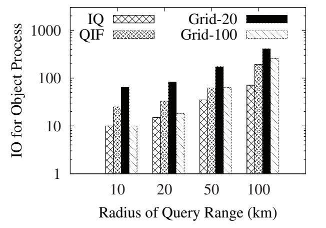


(a) Runtime for Object Process

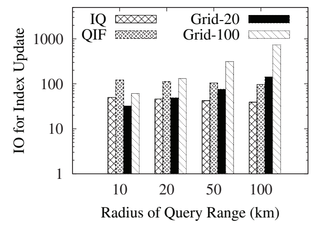


(b) I/O for Object Process

Figure 12: Effect of the Number of Object Keywords

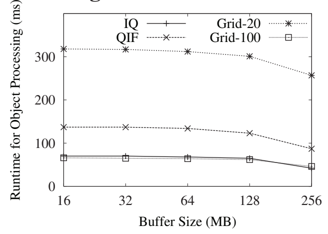


(a) I/O for Object Process

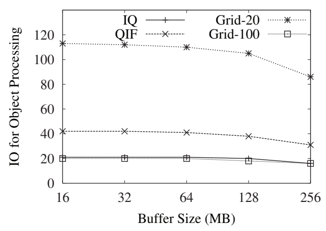


(b) I/O for Index Update

Figure 13: Effect of Query Range Size

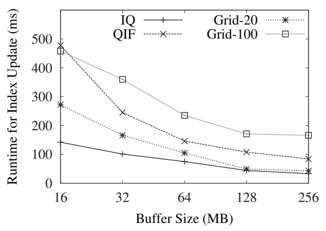


(a) Runtime for Object Process

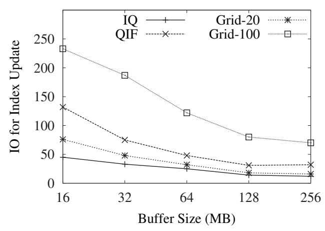


(b) I/O for Object Process

Figure 14: Effect of Buffer Size w.r.t Object Process



(a) Runtime for Index Update



(b) I/O for Index Update

Figure 15: Effect of Buffer Size w.r.t Index update

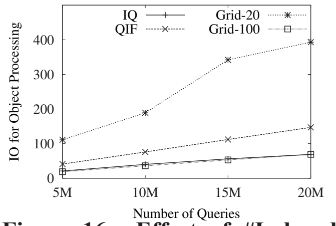


Figure 16: Effect of #Indexed Queries

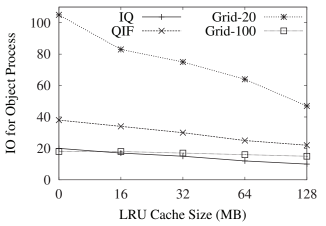


Figure 17: Effect of LRU Cache

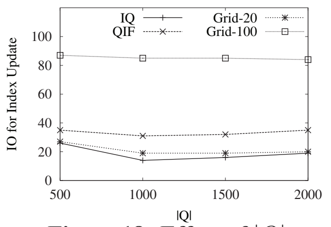


Figure 18: Effect of $|Q|$

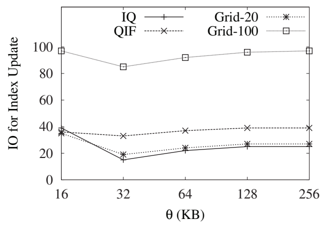


Figure 19: Effect of θ

semantics is splitted into multiple queries. Thus we need to store those queries into multiple inverted lists under each relevant node or cell. However, we only store a query of which keywords are connected by AND into a single inverted list of the least frequently keyword under each relevant node or cell.

A4 Effect of the Number of Object Keywords: Figure 12 shows that the object processing costs of all the indices increase linearly as we vary the number of of object keywords from 5 to 20. This is resulted from that for each object keyword, the pages in corresponding inverted list are to be retrieved. Obviously, index update cost is not affected here and the results are ignored.

A5 Effect of Query Range Size: This experiment evaluates the performance as we vary the radius of query range. Figure 13(b) shows that IQ-tree and QIF perform slightly better in index update when the query range is large. However, Grid-20 and Grid-100 exhibit a contrasting trend, they perform much better in index update when the query range is small. The reason is that Q-tree based indices tend to store queries with large range into a node with large spatial region. However, grid based indices have to use more cells to index queries with larger range, and this their performance drops.

For the same reason, IQ-tree and QIF exhibit a relatively constant performance in terms of object processing, while the performance of Grid-20 and Grid-100 deteriorate significantly as the query range size increases, which is indicated by Figure 13(a).

A6 Effect of the Buffer Size: This round of experiments evaluate the performance for both object processing and index update when we vary the buffer size. As shown in Figure 14, the object processing costs of all indices decrease when we increase the buffer size. The reason is that the index size is decreased as we increase the buffer size. Thus fewer pages are to be retrieved while processing objects.

Figure 15 shows that the cost of index update is higher with small buffer. When the buffer is full, we need to choose a batch of queries to write to the disk. Smaller buffer means that the index update operation will become more frequent, resulting in more I/O cost.

We also conduct experiments of varying the threshold for postings list, and we observe similar trend as varying the buffer size. The results are ignored due to the space limitation.

A7 Effect of the number of Indexed Queries: In this experiment, we vary the number of queries that are stored in disk to evaluate the scalability of each index in terms of object processing. Increasing the number of stored queries leads to the increase of page files in each inverted list. Consequently, more pages will be retrieved while processing objects.

Figure 16 shows that the object processing costs for all the four indices exhibit a linearly increasing trend as we vary the number of stored queries from 5M to 20M.

A8 Effect of LRU Cache Size: Figure 17 shows the I/O cost for object processing as we vary the cache size from 16MB to 128MB. As expected, extra cache improves the performance of all indices. When we increase the cache size, the improvements of Grid-20 and IQ-tree, whose space costs are lower, are more conspicuous than are Grid-100 and QIF. Consequently, the indices with lower space cost tend to render a more significant improvement on the I/O performance as the cache size mounts up.

A9 Effect of $|Q|$: This experiment is to study the effect of the number of queries in a batch. Figure 18 suggests that index update exhibits the best performance when $|Q|$ is around 1000. If $|Q|$ is too small, a large number of queries cannot form complete pages since queries are partitioned into different tree nodes or grid cells. But if Q is too large, the span of the expiration time for queries in a batch will be large, which will lead to inefficient page deletion.

A10 Effect of θ: This experiment studies the effect of the threshold θ, which indicates the size of individual postings list. Figure 19 suggests that index update exhibits the best performance when θ is 32KB. When θ is set at 16KB, index update will be conducted frequently even if the number of queries in the batch is smaller than $|Q|$. When θ is larger than 64KB, the average size of postings list in memory will become larger so that it may cost more time to select a batch of queries from corresponding in-memory postings list.

6. RELATED WORK

Spatial-keyword queries. Spatial-keyword queries return users service objects (e.g., shops, restaurants) that are close to the query location and whose text descriptions are relevant to the query keywords. The recent research efforts focus on efficiently processing spatial-keyword queries using geo-textual indices. Most geo-textual indices combine R-tree for spatial indexing and the inverted file for text indexing, such as the IR^2-tree [11], the IR-tree [8], the bR*-Tree [25], and the S2I [16]. Take the IR^2-tree for example, it augments an R-tree with signatures. The new hybrid index structure enables to utilize both spatial information and text information to prune search space at query time. These R-tree based index structures are not suitable for the problem studied in this paper since (1) it is difficult to employ these index structures to handle the stream of incoming BRC queries; and (2) they do not consider the temporal information of BRC queries.

Christoforaki et al. [7] propose SFC-QUAD that combines the space filling curve and inverted file. The SFC-QUAD index is essentially an inverted file in which the docIDs in each inverted list are ordered based on their spatial positions on the Z-curve. SFC-QUAD maintains a Quad-tree in memory so that the Z-curve order can be easily acquired simply by traversing the Quad-tree in a depth-first manner. This index is not suitable to our problem since there exits no sensible way to maintain the index in the presence of frequent updates needed by the stream of incoming BRC queries.

As discussed in Section 3.2, the spatial-first grid hybrid index (TS) [19] uses uniform grid cell to index the spatial information of geo-textual objects. TS maintains an inverted file for each cell. Due to its simplicity, it can be extended to handle our problem. We use the extended version as a baseline in our experimental study. In addition, Yao et al. [24] addresses the problem of approximate string search on geo-textual data. The collective spatial keyword search [3] is to find a group of objects that are close to a query point and collectively cover a set of a set of query keywords. Wu et al. [20] cover the problem of maintaining the result set of top-k spatial keyword queries while the user is moving.

Information Filtering and Selective Dissemination. Information Filtering and Selective Dissemination have been extensively investigated, e.g., [12]. Typically, user profiles are represented by a set of keywords, and a document is matched with the user profiles based on a boolean expression or similarity measure [4]. The Stanford Information Filtering Tool (SIFT) [22] is a keyword-based text filtering system for Internet News articles. Fabret et al. [10] propose an efficient algorithm for publish/subscribe systems, focusing on subscription consisting of conjunction of predicates. Recently, Mouratidis et al. [14] propose a new solution for processing continuous text queries efficiently.

Our work differs from these studies in that it is targeted at application domains where both queries and objects are geo-textual.

Continuous Queries. Our work is also related to the Continuous Queries in relational databases, which have been proposed for information delivery on the Internet (e.g., NiagaraCQ [5]). These systems focus on relational techniques. For example, NiagaraCQ groups query plans of continuous queries using common expression.

Continuous queries have also been studied extensively for spatial databases. They mainly focus on continuous spatial query processing on moving objects, e.g., the *continuous k nearest neighbor* query and *continuous range query*. The k-NN query finds the nearest k objects to a given query point among all the moving objects. The continuous range query returns the moving objects that are currently located inside the query region. Example algorithms include SINA [13], and CPM [15]. These continuous spatial queries are essentially different from the BRC queries.

Spatio-temporal indexing. The spatio-temporal indices mainly focus on indexing moving objects with a predictable trajectory, such as ST^2B-tree [6], RUM-tree [21], and TPR*-tree [18]. However, those indexing structures cannot be applied for storing BRC queries. Several techniques, e.g., SWST [17] and PIST [1], have been proposed to index discretely moving objects. However, those indices are not designed for indexing objects with a spatial region.

We are also aware of proposals that develop cost model of Quad-tree based index. For example, TrajStore [9] is a Quad-tree based index for storing trajectories. A cost model is proposed to dynamically determine the size of the spatial cells in quad-tree for storing trajectories. However, the motivation and indexing techniques used in the IQ-tree are significantly different from those in TrajStore, which is summarized in two aspects. (1) TrajStore aims at indexing trajectories, which is different from BRC queries. (2) TrajStore does not consider the text information, but the IQ-tree considers both text and spatial information while storing each BRC query.

7. CONCLUSIONS AND FUTURE WORK

This paper proposes a new problem of matching a stream of BRC queries over a stream of geo-textual objects in real time. We develop a new system for the problem. In particular, we propose the IQ-tree and novel cost models for managing a stream of incoming BRC queries. Based on the IQ-tree index, we also propose algorithms for matching the BRC queries with incoming geo-textual objects. Results of empirical studies demonstrate that the paper's proposals are capable of excellent performance. Results also show that the proposals significantly outperform two baseline methods.

This work opens to a number of promising directions for future work. First, it is of great interest to develop solutions in a distributed setting. Second, it is of interest to develop solutions for storing and indexing the stream of geo-textual objects within a sliding window so that historical geo-textual objects in the sliding window can be retrieved. This will be complementary and orthogonal to the techniques developed in this paper. Third, it would be of interest to consider other types of continuous queries, e.g., continuous top-k spatial-keyword queries.

8. ACKNOWLEDGEMENTS

This work is supported in part by a grant awarded by a Singapore MOE AcRF Tier 2 Grant (ARC30/12) and a Singapore MOE AcRF Tier 1 Grant (RG66/12).

9. REFERENCES

[1] V. Botea, D. Mallett, M. A. Nascimento, and J. Sander. Pist: An efficient and practical indexing technique for historical spatio-temporal point data. *GeoInformatica*, 12(2):143–168, 2008.

[2] X. Cao, L. Chen, G. Cong, C. S. Jensen, Q. Qu, A. Skovsgaard, D. Wu, and M. L. Yiu. Spatial keyword querying. In *ER*, pages 16–29, 2012.

[3] X. Cao, G. Cong, C. S. Jensen, and B. C. Ooi. Collective spatial keyword querying. In *SIGMOD*, pages 373–384, 2011.

[4] U. Çetintemel, M. J. Franklin, and C. L. Giles. Self-adaptive user profiles for large-scale data delivery. In *ICDE*, pages 622–633, 2000.

[5] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. Niagaracq: a scalable continuous query system for internet databases. In *SIGMOD*, pages 379–390, 2000.

[6] S. Chen, B. C. Ooi, K.-L. Tan, and M. A. Nascimento. St^2b-tree: a self-tunable spatio-temporal b^+-tree index for moving objects. In *SIGMOD*, pages 29–42, 2008.

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

[8] G. Cong, C. S. Jensen, and D. Wu. Efficient retrieval of the top-k most relevant spatial web objects. *PVLDB*, 2(1):337–348, 2009.

[9] P. Cudré-Mauroux, E. Wu, and S. Madden. Trajstore: An adaptive storage system for very large trajectory data sets. In *ICDE*, pages 109–120, 2010.

[10] F. Fabret, H. A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. In *SIGMOD*, pages 115–126, 2001.

[11] I. D. Felipe, V. Hristidis, and N. Rishe. Keyword search on spatial databases. In *ICDE*, pages 656–665, 2008.

[12] P. W. Foltz and S. T. Dumais. Personalized information delivery: an analysis of information filtering methods. *Commun. ACM*, 35(12):51–60, Dec. 1992.

[13] M. F. Mokbel, X. Xiong, and W. G. Aref. Sina: scalable incremental processing of continuous queries in spatio-temporal databases. In *SIGMOD*, pages 623–634, 2004.

[14] K. Mouratidis and H. Pang. Efficient evaluation of continuous text search queries. *IEEE Trans. Knowl. Data Eng.*, 23(10):1469–1482, 2011.

[15] K. Mouratidis, D. Papadias, and M. Hadjieleftheriou. Conceptual partitioning: an efficient method for continuous nearest neighbor monitoring. In *SIGMOD*, pages 634–645, 2005.

[16] J. B. Rocha-Junior, O. Gkorgkas, S. Jonassen, and K. Nørvåg. Efficient processing of top-k spatial keyword queries. In *SSTD*, pages 205–222, 2011.

[17] M. Singh, Q. Zhu, and H. V. Jagadish. Swst: A disk based index for sliding window spatio-temporal data. In *ICDE*, pages 342–353, 2012.

[18] Y. Tao, D. Papadias, and J. Sun. The tpr*-tree: An optimized spatio-temporal access method for predictive queries. In *VLDB*, pages 790–801, 2003.

[19] S. Vaid, C. B. Jones, H. Joho, and M. Sanderson. Spatio-textual indexing for geographical search on the web. In *SSTD*, pages 218–235, 2005.

[20] D. Wu, M. L. Yiu, C. S. Jensen, and G. Cong. Efficient continuously moving top-k spatial keyword query processing. In *ICDE*, pages 541–552, 2011.

[21] X. Xiong and W. G. Aref. R-trees with update memos. In *ICDE*, page 22, 2006.

[22] T. W. Yan and H. García-Molina. Index structures for selective dissemination of information under the boolean model. *ACM Trans. Database Syst.*, 19(2):332–364, June 1994.

[23] T. W. Yan and H. Garcia-Molina. Duplicate removal in information system dissemination. In *VLDB*, pages 66–77, 1995.

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

[25] 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*, pages 688–699, 2009.