Batch processing of Top-k Spatial-textual Queries

Farhana M. Choudhury J. Shane Culpepper Timos Sellis
School of CSIT, RMIT University, Melbourne, Australia
{farhana.choudhury,shane.culpepper,timos.sellis}@rmit.edu.au

ABSTRACT

Top-*k* spatial-textual queries have received significant attention in the research community. Several techniques to efficiently process this class of queries are now widely used in a variety of applications. However, the problem of how best to process multiple queries efficiently is not well understood. Applications relying on processing continuous streams of queries, and offline pre-processing of other queries could benefit from solutions to this problem. In this work, we study practical solutions to efficiently process a set of top-*k* spatial-textual queries. We propose an efficient best-first algorithm for the batch processing of top-*k* spatial-textual queries that promotes shared processing and reduced I/O in each query batch. By grouping similar queries and processing them simultaneously, we are able to demonstrate significant performance gains using publicly available datasets.

Categories and Subject Descriptors

D.2.8 [**Database applications**]: Spatial-textual databases

General Terms

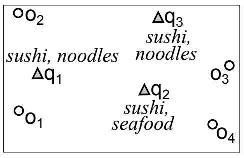
Algorithms, Experimentation, Performance

Keywords

Spatial-textual queries, Batch processing

1. INTRODUCTION

As a result of the increasing popularity of GPS enabled mobile devices, the volume of content associated with both a geographic location and a text description is growing rapidly on the web. Examples include points of interest such as stores or tourist attractions, social network posts, and location-based advertisements. In addition, the number of queries and the need to search content with local intent is also growing. For example, Google processes 5.7 billion searches per day, many of which are location-aware [1]. Another study shows that about 53% of Bing searches are geographical and have local intent [7].



Objects	Terms and frequency
o_1	$(sushi, 1)$
o_2	$(sushi, 1), noodles, 5)$
o_3	$(sushi, 5), (seafood, 5)$
o_4	$(seafood, 4), (noodles, 1)$

Figure 1: Example top-k spatial-textual queries

A *Top-k Spatial-Textual Query* is an important search task that has been extensively studied in the literature [3, 6, 8, 10]. Given a set of spatial-textual objects, a top-*k* spatial-textual query returns the *k* most similar objects to the query, where the similarity measure considers both spatial proximity and textual similarity. For example, consider the restaurants o_1, o_2, o_3, and o_4 shown in Figure 1. The table shows the terms and corresponding frequencies contained in the text description of each restaurant. Let q_1 be a top-1 spatial-textual query with a location as shown in Figure 1 and the query keywords are ("`sushi`", "`noodles`"). Even though restaurant o_1 is closer to q_1, o_2 is the highest ranking result when both spatial and textual similarity are considered for q_1.

In this paper, we study the problem of *Batch Processed Top-k Spatial-Textual Queries*. Our goal is to process a set of queries in a single pass. Consider the example in Figure 1 again. Let, q_1, q_2, and q_3 be three different top-*k* spatial-textual queries to be batch processed. Assume that $k = 1$ for q_1 and q_3, and $k = 2$ for q_2. The queries q_1 and q_3 both have the query keywords ("`sushi`", "`noodles`") and the query keywords of q_2 are "`sushi`" and "`seafood`". In this example, the result returned for q_1 is $\{o_2\}$, q_2 is $\{o_3, o_4\}$, and the result of q_3 is $\{o_3\}$ when taking both spatial and textual relevance into account.

Given a set of top-*k* spatial-textual queries, our aim is to compute the results for all the queries concurrently and efficiently. The main motivation of this problem is to improve the performance when processing a large number of queries together, especially if the results of the queries have large overlaps. An example application is to continuously process a stream of queries. At each time slot, the queries that have arrived can be processed as a batch, instead of processing individually. Consider another scenario where a large number of similar queries are issued by the users of the same area to get the news on a recent incident, for example, a road accident. The batch processing can be useful to optimize the overall performance by grouping such queries. Another application of the batch processing is as a pre-processing step for other query types. For example, a company might use a set of query logs to detect potential customers who found their products in prior searches. In this case, the results for each query in the log can be batch processed offline.

In related literature, Wu et al. [9] have studied a similar problem called *Joint Top-k Spatial Keyword Query Processing*. Example applications suggested by Wu et al. include multiple query optimization by grouping/partitioning a large set of queries, and privacy aware query support by hiding a query among multiple fake queries. These queries could also benefit from the approaches proposed in this work. However, Wu et al. consider only the *Boolean* top-k spatial-textual queries where a query retrieves k objects closest to the query location containing *all* of the query keywords. The authors proposed a new index structure, the *W-IR-tree*, which uses a frequency based partitioning of the object terms for index construction. This index structure and associated pruning strategies are applicable for the Boolean top-k query where a relevant object should contain all the query keywords, thereby, not directly extensible to queries that consider both spatial and textual similarity.

To the best of our knowledge, we are the first to propose and address the *Batch Processed Top-k Spatial-Textual Query* problem. Given a set of top-k spatial-textual queries, we propose an efficient algorithm that processes all of the queries concurrently using a best-first approach. The key idea of the algorithm is to share the computation and I/O cost among all queries. This algorithm is generic and applicable to any existing spatial-textual index structure that can answer individual top-k queries. In summary, the key contributions of this paper are as follows:

- We introduce a novel problem, *Batch Processed Top-k Spatial-Textual Queries* that is useful in many real-life applications.
- We propose an efficient best-first algorithm that greatly reduces the overall computational costs by sharing the processing and I/O costs among the queries.
- We conduct an extensive experimental study to show the efficiency of our approach using publicly available datasets.

The rest of paper is organized as follows: Section 2 gives an overview of the related work. Section 3 presents the problem definition. In Section 4, we describe our proposed algorithms. Finally, the experimental evaluation is shown in Section 5, and the paper is concluded in Section 6.

2. RELATED WORK

In this section we review relevant previous work on top-k spatial-textual queries and the joint processing of these queries.

2.1 Top-k spatial-textual queries

Given a set of spatial-textual objects, a top-k spatial-textual query returns a ranked list of k objects with the highest ranking scores, where the scoring function is a combination of the distance to the query location and the text relevance to the query keywords. Existing approaches depend primarily on variations in indexing data structures to answer a query efficiently [3, 6, 8, 10]. However, these methods are designed to answer queries individually. The computations and associated I/O costs are not shared among multiple queries, and queries are not processed concurrently. This leads to many disk reads and computations to be repeated, especially if the queries being processed are similar, i.e., the queries share a lot of keywords and are located close in space.

2.2 Joint top-k spatial-textual queries

Wu et al. [9] studied the joint top-k spatial-textual queries, where, given a set of queries, queries are processed jointly as a single query. They addressed the problem for only *Boolean* top-k queries, i.e., a query retrieves k objects according to the distance from the query location that contain *all* the query keywords. They introduce the *W-IR-tree* and a variant, the *W-IBR-tree*, along with the GROUP algorithm that can be used with the *W-IR-tree* as well as other existing indexes, e.g., the IR-tree, the CDIR-tree [3, 8], etc. However, most of the pruning techniques proposed in [9] are limited to Boolean top-k queries. Moreover, the authors use the minimum bounding rectangle (MBR) of all the queries to estimate an upper bound on the distance to the k-th nearest object. This technique can lead to many unnecessary retrievals when the query locations are very sparse. Therefore, we investigate other alternatives to joint processing of top-k spatial-textual queries in our work.

3. PROBLEM STATEMENT

Let D be a geo-textual dataset where each object $o \in D$ is defined as a pair $(o.l, o.d)$. Here, $o.l$ represents the spatial location and $o.d$ is the associated text description. Let $q = (q.l, q.d, q.k)$ a spatial-textual query where $q.l$ is the query location, $q.d$ is the set of query keywords and $q.k$ is the number of objects to be returned as result.

A *Top-k spatial-textual query* q returns a ranked list of $q.k$ most relevant spatial-textual objects from D according to a relevance function $STS(o,q)$. Furthermore, a spatial-textual object o is considered relevant to q, iff $o.d$ contains at least one term $t \in q.d$. The *Batch Processed Top-k Spatial-Textual Queries* problem is to process a set Q of such queries concurrently. In this paper we use the following ranking function:

$$STS(o,q) = \alpha \cdot SS(o.l, q.l) + (1-\alpha) \cdot (1 - TS(o.d, q.d)), \quad (1)$$

where $SS(o.l, q.l)$ is the spatial proximity between $q.l$ and $o.l$, $TS(o.d, q.d)$ is the textual similarity between the query keywords and $o.d$, and the preference parameter $\alpha \in [0,1]$ is used to define the importance of one measure relative to the other. The value of both measures are normalized within the range $[0,1]$. A lower value of $STS(o,q)$ indicates a higher relevance between o and q.

Spatial proximity. The spatial relevance between $q.l$ and $o.l$ is defined as: $SS(o.l,q.l) = \frac{dist(o.l,q.l)}{d_{max}}$, where $dist(o.l,q.l)$ is the Euclidean distance between $o.l$ and $q.l$, and d_{max} is the maximum Euclidean distance between any two points in D.

Textual relevance. The text description $o.d$ is represented by a vector where each dimension corresponds to a distinct term in the document. The value of a term t in $o.d$ is computed using the language model [5] as:

$$\hat{p}(t|\theta_{o.d}) = (1-\lambda)\frac{tf(t,o.d)}{|o.d|} + \lambda\frac{tf(t,C)}{|C|}, \quad (2)$$

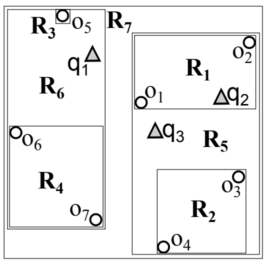
where $tf(t,o.d)$ is the number of occurrences of the term t in $o.d$, $tf(t,o.d)/|o.d|$ is the maximum likelihood estimate of term t in $o.d$; C is the concatenation of all documents in the collection, and λ is a smoothing parameter for Jelinek-Mercer smoothing. In the language model, the text relevance of an object o with respect to a query q is, $TS(o.d,q.d) = \frac{\sum_{t \in q.d} \hat{p}(t|\theta_{o.d})}{maxP}$, where $maxP$ is used to normalize the score into the range from 0 to 1 and is computed as, $maxP = \sum_{t \in q.d} \max_{o' \in D} \hat{p}(t|\theta_{o'.d})$.

4. PROPOSED APPROACH

We assume all objects $o \in O$ are stored on disk and indexed using a spatial-textual index. We use the IR-tree [3] to index the objects. A brief overview of the IR-tree is presented in Section 4.1 and a baseline approach is described in Section 4.2. In Section 4.3, the proposed algorithm for batch processing top-k spatial-textual queries is presented. In addition, we discuss the effect of different retrieval orders of the nodes of the tree in Section 4.4. The algorithms are generic and independent of the indexing method used.

4.1 The IR-tree

An IR-tree is an R-tree [4] where each node is augmented with a reference to an inverted file [11] for the documents in the subtree.



Objects	Terms and frequency
o_1	$(t_1,1),(t_2,4)$
o_2	$(t_4,1)$
o_3	$(t_1,5),(t_3,5)$
o_4	$(t_4,2)$
o_5	$(t_1,4),(t_2,1)$
o_6	$(t_1,1),(t_3,1)$
o_7	$(t_1,2),(t_4,3)$

Figure 2: A dataset of spatial-textual objects

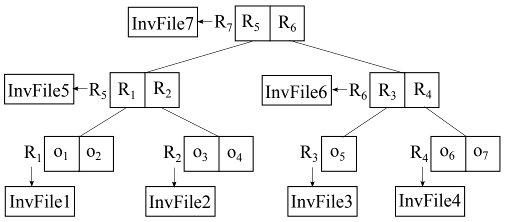


Figure 3: The example IR-tree tree

Each node R contains a number of entries of the form $(cp, rect, cp.di)$. If R is a leaf node, cp is the reference of an object $o \in D$, $rect$ is the bounding rectangle of o, and $o.di$ is an identifier of the text description. If R is a non-leaf node, then cp refers to a child node of R, $rect$ is the MBR of all entries of the child node, and $cp.di$ is the identifier of a pseudo text description. The pseudo text description is the union of all text descriptions in the entries of the child node. The weight of a term t in the pseudo-document is the maximum weight of the weights of this term in the documents contained in the subtree. Each node has a reference to an inverted file for the entries stored in the node. A posting list of a term t in the inverted file is a sequence of pairs $\langle d, w_{d,t}\rangle$, where d is the document id containing t, and $w_{d,t}$ is the weight of term t in d.

Figure 2 shows the locations and the text descriptions of an example dataset $O = \{o_1, o_2, \ldots, o_7\}$ and Figure 3 illustrates the IR-tree for O. Table 1 and Table 2 present the inverted files of the leaf nodes (InvFile 1 - InvFile 4) and the non-leaf nodes (InvFile 5 - InvFile 7), respectively. In this example, the weights are shown as term-frequencies to simplify the presentation, but the actual weights stored in the inverted files are pre-computed using Equation 2.

Table 1: Posting lists of the leaf nodes of example IR-tree

Term	InvFile 1	InvFile 2	InvFile 3	InvFile 4
t_1	$(o_1,1)$	$(o_3,5)$	$(o_5,4)$	$(o_6,1),(o_7,2)$
t_2	$(o_1,4)$		$(o_5,1)$	
t_3		$(o_3,5)$		$(o_6,1)$
t_4	$(o_2,1)$	$(o_4,2)$		$(o_7,3)$

Table 2: Posting lists of the non-leaf nodes of example IR-tree

Term	InvFile 5	InvFile 6	InvFile 7
t_1	$(R_1,1)$, $(R_2,5)$	$(R_3,4)$, $(R_4,2)$	$(R_5,5),(R_6,4)$
t_2	$(R_1,4)$	$(R_3,1)$	$(R_5,4),(R_6,1)$
t_3	$(R_2,5)$	$(R_4,1)$	$(R_5,5),(R_6,1)$
t_4	$(R_1,1),(R_2,2)$	$(R_4,3)$	$(R_5,2),(R_6,3)$

4.2 Baseline approach

The baseline for batch processing of top-k spatial-textual queries utilizes an existing algorithm [3] that processes a single query at a time. As a batch processing problem consists of a set Q of top-k spatial-textual queries, the baseline algorithm simply processes the queries $q \in Q$ individually.

The inputs of the baseline are a set of queries Q and a spatial-textual index (in our example, an IR-tree) over the set of objects O. The pseudo-code of the baseline is shown in Algorithm 1. When processing a query $q \in Q$, a best-first traversal algorithm is used to retrieve the top-k objects. The algorithm uses a priority queue PQ to maintain the objects and the nodes that are yet to be visited (line 1.3). The value of $STS(o,q)$ is used to order PQ. In each iteration, the top node E of PQ (the most relevance node) is dequeued (line 1.6). If E is an object, E is included in the result for q (lines 1.7-1.8). The process for a query q terminates when $q.k$ objects have been found (lines 1.9-1.10). Otherwise, the elements of E are read from the disk and the elements that contain at least one query keyword from $q.d$ are added to the queue (lines 1.12-1.14). Finally the algorithm returns the results for all of the queries in Q.

Algorithm 1: *Baseline approach(Q,IRtree)*

```
1.1  Initialize an array $H$ of $|Q|$ min-priority queues to order the top-$k$
     results for each query.
1.2  for each $q \in Q$ do
1.3      Initialize a min-priority queue $PQ$.
1.4      $PQ \leftarrow$ ENQUEUE($IRtree(root)$, 0)
1.5      while PQ not empty do
1.6          $E \leftarrow$ DEQUEUE($PQ$)
1.7          if $E$ is an object then
1.8              $H_q \leftarrow$ ENQUEUE($E, STS(E,q)$)
1.9              if SIZEOF($H_q$) $\geq q.k$ then
1.10                 break
1.11         else
1.12             for each element $e \in E$ do
1.13                 if $q.d \cap e.d \neq \varnothing$ then
1.14                     $PQ \leftarrow$ ENQUEUE($e, STS(e,q)$)
1.15 return $H$
```

A trace of processing required for the baseline is sketched in Table 3 where the data from Figure 2 for the queries q_1, q_2, and q_3 are used. Here, $\alpha = 0.5$, $q_1.d = \{t_2, t_3\}$, $q_1.k = 1$; $q_2.d = \{t_1\}$, $q_2.k = 2$; and $q_3.d = \{t_1, t_3\}$, $q_3.k = 1$.

Table 3: The example steps of the baseline approach

Query	E	PQ	H_q
q_1	R_7	$(R_6,0.2),(R_5,0.4)$	$-$
	R_6	$(R_3,0.3),(R_4,0.4),(R_5,0.4)$	$-$
	R_3	$(o_5,0.3),(R_4,0.4),(R_5,0.4)$	$-$
	o_5	$(R_4,0.4),(R_5,0.4)$	o_5
q_2	R_7	$(R_5,0.1),(R_6,0.5)$	$-$
	R_5	$(R_1,0.2),(R_2,0.3),(R_6,0.5)$	$-$
	R_1	$(o_1,0.2),(R_2,0.3),(R_6,0.5)$	$-$
	o_1	$(R_2,0.3),(R_6,0.5)$	o_1
	R_2	$(o_3,0.4),(R_6,0.5)$	o_1
	o_3	$(R_6,0.5)$	o_1,o_3
q_3	R_7	$(R_5,0.1),(R_6,0.6)$	$-$
	R_5	$(R_1,0.2),(R_2,0.25),(R_6,0.6)$	$-$
	R_1	$(R_2,0.25),(o_1,0.3),(R_6,0.6)$	$-$
	R_2	$(o_3,0.26),(o_1,0.3),(R_6,0.6)$	$-$
	o_3	$(o_1,0.3),(R_6,0.6)$	o_3

In the baseline, a node of the tree might be retrieved multiple times for different queries, which can result in high I/O costs. For instance, the nodes R_1, R_2, R_5 and the objects o_1, o_3 all are retrieved twice from the disk in this example. To overcome this drawback, we propose an efficient solution using batch processing of the queries, where a node is guaranteed to be retrieved at most once.

4.3 The batch processing algorithm

In this approach, if multiple queries require the retrieval of the same node, that node is guaranteed to be retrieved from disk at most once during the process. The pseudo-code is shown in Algorithm 2.

In order to process all of the queries in a single pass, two arrays of priority queues of size $|Q|$ are necessary. First, a min-priority queue H_q is maintained for each query $q \in Q$ to store the top-k results (line 2.1). For each query $q \in Q$, we also maintain a min-priority queue PQ_q to track the relevant nodes and the objects, where the key is the corresponding relevance score for q. The algorithm will continue to execute as long as the set Q is non-empty.

In each iteration, a priority queue PQ_r is selected at random and the top element E of that queue is processed (lines 2.6-2.7). We explain the reasoning behind this selection method and discuss the effect of other selection orders in Section 4.4. If E is an object, the top element o of PQ_q is dequeued and inserted into H_q as long as o is an object and H_q has less than $q.k$ elements for each PQ_q (lines 2.8-2.12). In this manner, all the queries that have any object including E in the top of their queues are considered. If PQ_q is empty or H_q has $q.k$ elements then q is marked as finished, and discarded from further computation (lines 2.13-2.14).

If E is not an object, then the elements of E are read from disk. For each query $q \in Q$, if the corresponding PQ_q contains E, the elements of E that have at least one keyword of $q.d$ are enqueued in PQ_q. The node E is then removed from these queues (lines 2.16-2.22). The process terminates when the results for all of the queries in Q are found. Finally, the results represented as an array of priority queues H is returned.

Algorithm 2: *Batch processing Top-k(Q,IRtree)*

```
2.1  Initialize an array $H$ of $|Q|$ min-priority queues to order the top-$k$
     results for each query.
2.2  Initialize an array $PQ$ of $|Q|$ min-priority queues to track node
     traversal for each query.
2.3  for each $q \in Q$ do
2.4      $PQ_q \leftarrow$ ENQUEUE(IRtree(root), 0)
2.5  while $Q$ not empty do
2.6      Select $PQ_r$ randomly.
2.7      $E \leftarrow$ TOP($PQ_r$)
2.8      if $E$ is an object then
2.9          for each $q \in Q$ do
2.10             while $PQ_q$ not empty and SIZEOF($H_q$) $< q.k$ and
                 TOP($PQ_q$) is an object do
2.11                 $o \leftarrow$ DEQUEUE($PQ_q$)
2.12                 $H_q \leftarrow$ ENQUEUE($o, STS(o,q)$)
2.13                 if $PQ_q$ is empty or SIZEOF($H_q$) $\geq q.k$ then
2.14                     Mark $q$ finished.
2.15     else
2.16         READ($E$)
2.17         for each $q \in Q$ do
2.18             if $E \in PQ_q$ then
2.19                 for each element $e \in E$ do
2.20                     if $q.d \cap e.d \neq \varnothing$ then
2.21                         $PQ_q \leftarrow$ ENQUEUE($e, STS(e,q)$)
2.22                 Remove $E$ from $PQ_q$.
2.23 Return $H$
```

Using other index structures. The key idea of the algorithm is share the I/Os and processing among queries. When an object is retrieved from the disk for a query, the score of the object is updated for all the queries that share this object in their corresponding queues. The other processing steps of the Algorithm 2 are same as the processing of a single query. Therefore, this algorithm is easy to extend for other index structures.

4.4 Selection order of the nodes

In this section we discuss the selection order of the tree nodes in each iteration of our algorithm.

Arbitrary selection. Recall that for each query q, the corresponding priority queue PQ_q is sorted according to the maximum relevance of nodes. Although in each iteration, the top element E is dequeued for a random query, all of the queries that have any object including E in the top of their queues are considered in lines 2.10-2.14 in the same iteration. Thus, we are applying a best-first approach not only for the query q_r, but also for other queries concurrently in each iteration. Moreover, only those nodes and objects are retrieved that are also required for individual processing, and a node or an object is retrieved only once.

In the best-first approach, the computation for a query q can be safely terminated iff $q.k$ objects are found or PQ_q is empty. The lines 2.13-2.14 ensure the terminating conditions for all queries regardless of the selection of E. Therefore, the algorithm is actually *independent of the retrieval order of the nodes*. We explain this further by contrasting with another retrieval order in the following.

Selecting the node that is the top one in the maximum number of queues. Let the node E that is the top element of the maximum number of queues be selected in each iteration. According to lines 2.10-2.12, if E is an object, then any object including E that is in the top of any queue is checked for being a result object of the corresponding query. If E is not an object, then the elements of E are enqueued in PQ_q if they have at least one keyword of $q.d$ and PQ_q contains E. In both conditions, all of the computations are the same as selecting E randomly. The computations do not rely on the selection of E.

Moreover, keeping track of the node that is the top of the maximum number of queues in each iteration may require some extra computation. Therefore, the random selection of a node is preferable. Table 4 demonstrates the execution of Algorithm 2 for the data from Figure 2. Note that the total number of iterations will be the same for any order of selecting E. As shown in the table, the priority queues of the queries are initialized with the root node of the tree. Consider iteration 2 as an instance where node R_5 is selected. Node R_5 is present in the priority queues of the queries q_2 and q_3, so the elements of R_5 – R_1 and R_2 – are retrieved from disk. Node R_1 and R_2 are enqueued in the priority queues of q_2 and q_3 along with the corresponding relevance scores. Then, node R_5 is removed from these queues.

5. EXPERIMENTAL EVALUATION

In this section, we present our results on evaluating the performance of our proposed algorithm for *Batch Processed Top-k Spatial-Textual Queries* and compare it with the baseline approach. All indices and algorithms are implemented in Java. The experiments were ran on a 24 core Intel Xeon $E5-2630$ running at 2.3 GHz using 256 GB of RAM, and 1TB 6G SAS 7.2K rpm SFF (2.5-inch) SC Midline disk drives. The Java Virtual Machine Heap size was set to 4 GB. All index structures are disk resident, and the page size was fixed at 4 kB.

5.1 Dataset and queries

All experiments are conducted using the Yahoo I3 Flickr dataset [1]. A total of 1 million image tags that are geo-tagged and contain at least one user specified tag were extracted from the collection and used for the experiments. The locations and tags are used as the location and text description of the spatial-textual objects in our problem. Table 5 lists the properties of the dataset. To evaluate scalability, three additional datasets of 2 million, 4 million, and 8 million were extracted from the collection, and are used where the spatial and term distribution of the objects are almost the same.

We generate the query sets using the dataset as follows. First, an object is picked randomly from the dataset and the location of the

[1] `http://webscope.sandbox.yahoo.com/catalog.php?datatype=i&did=67`

Table 4: The example steps of the batch processing approach

Iteration	E	PQ_q	H_q
		$(R_7, 0.0)$	—
		$(R_7, 0.0)$	—
		$(R_7, 0.0)$	—
1	R_7	$(R_6, 0.2), (R_5, 0.4)$	—
		$(R_5, 0.1), (R_6, 0.5)$	—
		$(R_5, 0.1), (R_6, 0.6)$	—
2	R_5	$(R_6, 0.2), (R_1, 0.5), (R_2, 0.6)$	—
		$(R_1, 0.2), (R_2, 0.3), (R_6, 0.5)$	—
		$(R_1, 0.2), (R_2, 0.25), (R_6, 0.6)$	—
3	R_1	$(R_6, 0.2), (o_1, 0.6), (R_2, 0.6)$	—
		$(o_1, 0.25), (R_2, 0.3), (R_6, 0.5)$	—
		$(R_2, 0.25), (o_1, 0.3), (R_6, 0.6)$	—
5	o_1	$(R_6, 0.2), (o_1, 0.6), (R_2, 0.6)$	—
		$(R_2, 0.3), (R_6, 0.5)$	o_1
		$(R_2, 0.25), (o_1, 0.3), (R_6, 0.6)$	—
6	R_2	$(R_6, 0.2), (o_1, 0.6), (o_3, 0.7)$	—
		$(o_3, 0.4), (R_6, 0.5)$	o_1
		$(o_3, 0.26), (o_1, 0.3), (R_6, 0.6)$	—
7	o_3	$(R_6, 0.2), (o_1, 0.6), (o_3, 0.7)$	—
		$(R_6, 0.5)$	o_1, o_3
		$(o_1, 0.3), (R_6, 0.6)$	o_3
8	R_6	$(R_3, 0.3), (R_4, 0.4), (o_1, 0.6), (o_3, 0.7)$	—
		-	o_1, o_3
		-	o_3
9	R_3	$(o_5, 0.3), (R_4, 0.4), (o_1, 0.6), (o_3, 0.7)$	—
		-	o_1, o_3
		-	o_3
10	o_5	-	o_5
		-	o_1, o_3
		-	o_3

Table 5: Description of dataset

Property	**Cardinality**
Total number of objects	1,000,000
Total number of unique terms	166,317
Average number of unique terms per object	6.9
Mode number of unique terms per object	3
Total number of terms in dataset	6,936,385

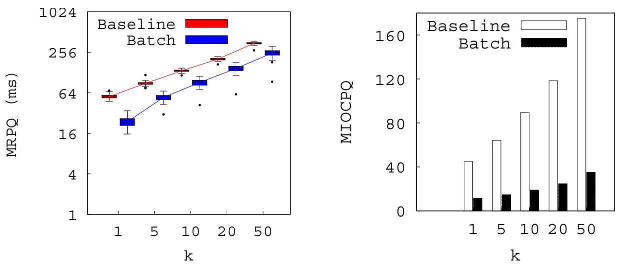
object is used as the query location. Then, a pre-defined number of words are randomly selected from the object as the query keywords. A pre-defined number of these queries are created to be the set of input queries Q for the batch processing problem. The default number of queries in a batch is 100. In this work, we generate 100 such batches and report the average performance.

5.2 Performance Evaluation

In this section, we evaluate the performance of the baseline and our proposed approach described in Section 4.3 by varying the parameters listed in Table 6. In all experiments, we employ the default settings to study the impact on: (i) the mean I/O cost (number of pages accessed) per query (MIOCPQ) of a batch; and (ii) the mean runtime per query (MRPQ) of a batch, while varying a single parameter. As multiple layers of cache exist between a Java application and the physical disk, we report simulated I/O costs in the experiments instead of physical disk I/Os. The number of simulated I/Os is increased by 1 when a node of the tree is visited. When an inverted file is loaded, the number of simulated I/Os is increased by the number of blocks (4 kB per block) for storing the list.

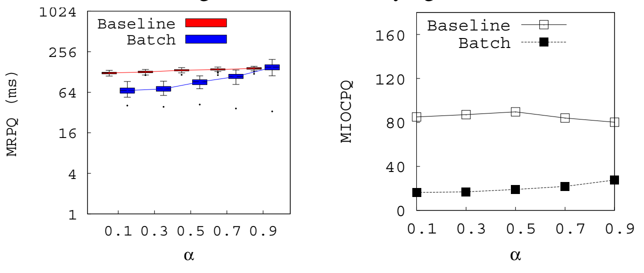
Table 6: Parameters

Parameter	**Range**	**Default**
k	$1, 5, 10, 20, 50$	10
α	$0.1, 0.3, 0.5, 0.7, 0.9$	0.5
$\lvert Q \rvert$	$100, 200, 400, 800, 1600$	100
No. of query keywords	$1, 2, 3, 4, 5, 6$	3
No. of unique query keywords, w	$5, 10, 20, 50, 100$	20
No. of objects	1M, 2M, 4M, 8M	1M



(a) Runtime (b) I/O cost

Figure 4: Effect of varying k



(a) Runtime (b) I/O cost

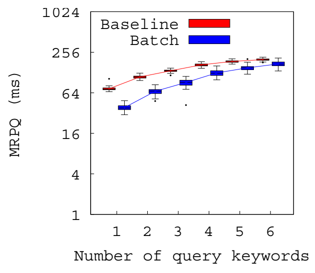
Figure 5: Effect of varying α

In the experiments, the performance is evaluated using cold queries. The runtimes for multiple runs are shown as boxplots, where each boxplot summarizes the values as follows: the solid line indicates the median; the bounding box shows the first and third quartiles; the whiskers show the range, up to 1.5 times of the interquartile range; and the outliers beyond this value are shown as separate points. The average values are shown as connecting lines.

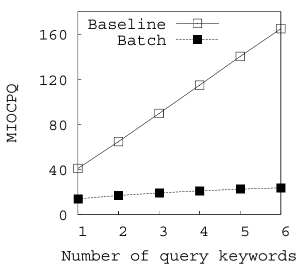
Varying *k*. In this experiment, we vary the value of k and investigate the effect. The result is shown in Figure 4. Here, both the runtime and the I/O cost for each method increase as k increases. Since the batch processing algorithm visits the disk pages only once if they are shared among the queries, the cost of our proposed method is significantly lower than the baseline. Furthermore, the advantage of the batch processing algorithm increases when k increases. The height of the boxplots indicate that the runtime of multiple runs do not differ much from one another.

Varying α. Figure 5 reports the result of the experiment where we study the effect of varying α. A higher value of α indicates more preference to spatial similarity. As the location of queries in a batch is expected to be sparse, a higher value of α leads to fewer shared IR-tree nodes. Therefore, the cost of batch processing increases as α increases. On average our algorithm requires 4 times less I/O than the baseline.

Varying the number of keywords per query. We vary the number of keywords per query in a batch from 1 to 6 and investigate the effect on performance in Figure 6. The mean I/O cost per query (MIOCPQ) and the mean runtime per query (MRPQ) in the baseline method increases proportionally with the increase in the number of keywords per query in a batch, as more nodes become relevant to these queries. In contrast, the I/O cost in our proposed algorithm remains almost constant as a node is retrieved at most once and processed for all queries concurrently. The runtime of our al-

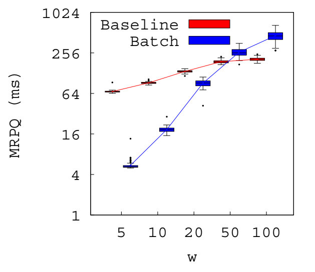


(a) Runtime

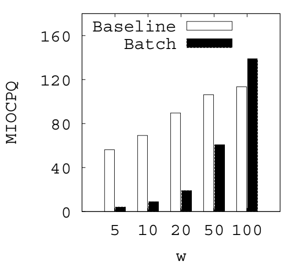


(b) I/O cost

Figure 6: Effect of varying the number of keywords per query



(a) Runtime



(b) I/O cost

Figure 7: Effect of varying *w*

gorithm increases with the increase in the number of keywords per query though, as the number of similarity computations increases with number of query keywords. Our algorithm outperforms the baseline for both metrics, where the difference in performance is higher for higher number of query keywords.

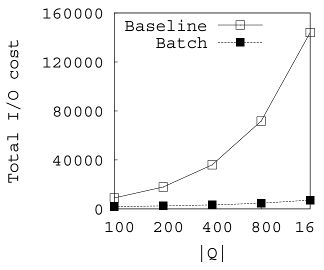
Varying the number of unique query keywords in a batch (w). Figure 7 shows the effect on performance for varying the total number w of unique query keywords in a batch. A lower value indicates the queries share more common keywords. As shown in Figure 7, the costs increase with the increase of w for both methods. Our proposed method outperforms the baseline when there is a high overlap among the queries in a batch. The baseline starts to outperform in terms of I/O cost and runtime when w is greater than approximately 70 and 40, respectively.

Scalability. We vary the number of queries $|Q|$ in a batch, and the total number of objects in the dataset to evaluate scalability. Figure 8a shows the effect of varying $|Q|$ on the total I/O cost of processing a batch of queries. As the number of queries increases, the cost of the baseline increases proportionally as it processes the queries one by one. Although the cost of our algorithm increases as the number of queries increases, it is significantly less than the cost of the baseline as it visits each node at most once.

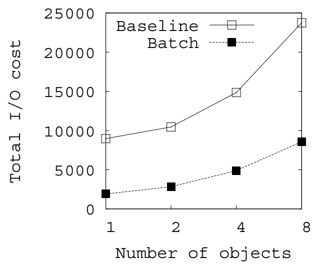
Figure 8b shows the performance when varying the number of objects in the dataset. The total I/O cost of both methods increase with the increase in the number of objects as more nodes are required to be retrieved. The baseline requires approximately 2 times more I/O than our proposed approach. The effect on runtime shows similar trend for varying both parameters.

6. CONCLUSION

This paper explored batch processing algorithms for efficient processing of top-k spatial-textual queries. The problem has several real-life applications. We have proposed a baseline derived from state-of-the-art approaches to single query processing, and extended the approach to support efficient batch processing. Our algorithm can improve the overall efficiency of batch processing significantly by sharing processing and I/O costs of the queries.



(a) Effect of varying $|Q|$



(b) Effect of varying the number of objects

Figure 8: Scalability

Through extensive experiments using publicly available datasets, we show that our approach outperforms the baseline in terms of I/O costs and total runtime. Our approach is particularly amenable to queries which share a large number of keywords, and/or are in close proximity to each other. In future work, we will explore alternative algorithms to minimize computational costs in groups of queries, and explore efficient approaches to group similar queries into batches in realtime.

7. ACKNOWLEDGMENTS

This work was supported by the Australian Research Council's *Discovery Projects* Scheme (DP140101587). Shane Culpepper is the recipient of an Australian Research Council DECRA Research Fellowship (DE140100275). Farhana Choudhury is the recipient of a scholarship from National ICT Australia. We thank Lisi Chen, Gao Cong, Christian S. Jensen, and Dingming Wu for providing the implementation of the IR-tree in [2].

References

[1] Google annual search statistics. `http://www.statisticbrain.com/google-searches`. [Online; accessed 20-03-2015].

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

[3] 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.

[4] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.

[5] J. M. Ponte and W. B. Croft. A language modeling approach to information retrieval. In *SIGIR*, pages 275–281, 1998.

[6] 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.

[7] SearchEngineLand. Microsoft: 53 percent of mobile searches have local intent. `http://searchengineland.com/microsoft-53-percent-of-mobile-searches-have-local-intent-55556`. [Online; accessed 20-03-2015].

[8] D. Wu, G. Cong, and C. S. Jensen. A framework for efficient spatial web object retrieval. *PVLDB*, 21(6):797–822, 2012.

[9] D. Wu, M. L. Yiu, G. Cong, and C. S. Jensen. Joint top-k spatial keyword query processing. *TKDE*, 24(10):1889–1903, 2012.

[10] L. Zhisheng, K. C. K. Lee, Z. Baihua, L. Wang-Chien, L. Dik Lun, and W. Xufa. IR-tree: An efficient index for geographic document search. *TKDE*, 23(4):585–599, 2011.

[11] J. Zobel, A. Moffat, and K. Ramamohanarao. Inverted files versus signature files for text indexing. *ACM Trans. Database Syst.*, 23(4): 453–490, 1998.