Supporting Location-Based Approximate-Keyword Queries

Sattam Alsubaiee
University of California, Irvine
salsubai@ics.uci.edu

Alexander Behm
University of California, Irvine
abehm@ics.uci.edu

Chen Li
University of California, Irvine
chenli@ics.uci.edu

ABSTRACT

Many Web sites support keyword search on their spatial data, such as business listings and photos. In these systems, inconsistencies and errors can exist in both queries and the data. To bridge the gap between queries and data, it is important to support approximate keyword search on spatial data. In this paper we study how to answer such queries efficiently. We focus on a natural index structure that augments a tree-based spatial index with capabilities for approximate keyword search. We systematically study how to efficiently combine these two types of indexes, and how to search the resulting index to find answers. We develop three algorithms for constructing the index, successively improving the time and space efficiency by exploiting the textual and spatial properties of the data. We experimentally demonstrate the efficiency of our techniques on real, large datasets.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications—*spatial databases and GIS*; H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing

General Terms

Algorithms, Performance

Keywords

Approximate Keyword Search, Spatial Data

1. INTRODUCTION

An increasing number of Web sites support location-based keyword search on their data objects such as business listings and photos. They accept queries consisting of two conditions: a set of keywords and a spatial location. The goal is to find objects with these keywords close to the location. Such a query is called a *spatial-keyword* query [10]. For instance, there are several local-search Web sites, such as Bing Maps, Google Maps, Yahoo! Local, and Yelp.

In these systems, inconsistencies and errors can exist in user queries and data. For instance, a user might be looking for a restaurant called `Alouette` close to `New York`. The following is the corresponding query:

$$Q_1\text{: } (\texttt{Alouette}) \textit{ near } (\texttt{New York}).$$

The Web site returns listings close to `New York` that have the keyword `Alouette`. Sometimes, users may not know the exact spelling of the entities they are looking for. For instance, a user could have heard of the restaurant by word-of-mouth but is unfamiliar with its exact spelling, and issues the following query with a typo:

$$Q'_1\text{: } (\texttt{Aloette}) \textit{ near } (\texttt{New York}).$$

Errors can exist in data as well. For instance, Flickr supports location-based photo search.[1] Consider a query that asks for photos of `Alcatraz` close to `San Francisco`:

$$Q_2\text{: } (\texttt{Alcatraz}) \textit{ near } (\texttt{San Francisco}).$$

Since Flickr photos are manually uploaded and tagged by users, the title or the description of a photo may have spelling errors. Query Q_2 may not be able to find a photo with the mistyped title `Alkatraz`.

Finding relevant answers to such queries is important. Unfortunately, most existing location-based search engines could not provide correct answers to the query Q'_1 even with a single typo (as of July 1, 2010). One simple solution to this problem on a spatial dataset is to build a collection of keywords from the dataset. For a mistyped keyword, we find words from the collection that are similar to the keyword. We use these similar keywords to suggest another query or find objects with these keywords. The main drawback of this approach is that it does not consider the location condition when relaxing the mistyped keyword.

Contributions: In this paper, we study how to support approximate keyword search on spatial data. Answering such queries efficiently is critical to these Web sites to achieve a high query throughput to serve many concurrent users. Although there are studies on both approximate keyword search and location-based search, the problem of supporting both types of search simultaneously has received little attention. To answer such queries, a natural index structure is to augment a tree-based spatial index with approximate-string indexes such as a gram-based inverted index or a trie-based index. The main focus of this work is a systematic study on how to efficiently combine

[1]http://www.flickr.com/map

these two types of indexes, and how to search the resulting index (called LBAK-tree) to find answers.

We develop three algorithms in this context. First, in Section 3, we show a basic algorithm that selects a fixed level of nodes in the spatial tree to store approximate-string indexes. (A brief description of a system prototypes using this approach is presented in [1].) Second, in Section 4, we develop a cost-based algorithm that judiciously selects a set of nodes in the tree to store approximate indexes. Our cost model utilizes the spatial distribution of objects within each node to build the index structure with a space budget. Third, in Section 5 we continue improving the solution by exploiting the frequency distribution of keywords. We show how to further reduce the index size without sacrificing query time. We have conducted a comprehensive experimental study to evaluate the proposed techniques. Our results (Section 6) show the efficiency and scalability of these optimizations.

1.1 Related Work

Spatial keyword search: There are several studies on answering spatial queries for exact matching of keywords [16, 13, 5, 10, 7, 6, 15]. Chen et al. [5] used separate indices for the spatial and textual information. Zhou et al. [16] suggested a hybrid index that combines spatial and inverted indexes. They use an R*-tree [2] and build inverted indexes for the keywords at the leaf nodes; or use an inverted index to store the keywords and build an R*-tree for each keyword. These techniques do not simultaneously use the spatial and the textual information for pruning. Other studies [10, 7, 6, 15] used the textual and the spatial information conjunctively. They augment a tree-based spatial index with textual information in each node. Our work complements these studies by allowing approximate keyword matching.

Approximate string search: We refer to the problem of conjunctive keyword search with relaxed keywords as *approximate keyword search*. An important subproblem of approximate *keyword* search is that of approximate *string* search, defined as follows. Given a collection of strings, find those that are similar to a given query string. There are two main families of approaches to answer such queries. (1) Trie-based method: The string collection is stored as a trie (or a suffix tree). To answer an approximate string query, we traverse the trie and prune subtrees that cannot be similar to the query. Popular pruning techniques use an NFA or a dynamic programming matrix with backtracking. We refer the reader to [12] for more details. (2) Inverted-index method: Its main idea is to decompose each string in the collection to small overlapping substrings (called grams) and build an inverted index on these grams. More details on fast indexing and search algorithms can be found in [9, 11, 12].

Spatial approximate-keyword search: Yao et al. [14] proposed a structure called MHR-tree to answer spatial approximate-keyword queries. They enhance an R-tree [8] with min-wise signatures at each node to compactly represent the union of the grams contained in objects of that subtree. They then use the concept of set resemblance between the signatures and the query strings to prune branches in the tree. The main advantage of this approach is that its index size does not require a lot of space since the min-wise signatures are very small. However, the method could miss query answers due to the probabilistic nature of the signatures, while our approach can guarantee to find all true answers. In Section 6, we compare these two solutions experimentally.

2. PROBLEM FORMULATION

The problem of approximate keyword search on spatial data can be formulated as follows. Consider a collection of spatial objects $o_1, \ldots, o_n$, each having a textual description (a set of keywords) and a location. A spatial approximate-keyword query $Q = \langle Q_s, Q_t \rangle$ consists of two conditions: a spatial condition Q_s such as a rectangle or a circle, and an approximate keyword condition Q_t having a set of k pairs $\{\langle w_1, \delta_1 \rangle, \langle w_2, \delta_2 \rangle, \ldots, \langle w_k, \delta_k \rangle\}$, each representing a keyword w_i with an associated similarity threshold δ_i. The similarity thresholds refer to a similarity measure such as edit distance, Jaccard, etc., which could be different for each keyword. Our goal is to find all objects in the collection that are within the spatial region Q_s and satisfy the approximate keyword condition Q_t. We focus on conjunctive approximate keyword queries; thus, an object satisfies the approximate keyword condition if for each keyword w_i in Q_t, the object has a keyword in its description whose similarity to w_i is within the corresponding threshold δ_i. We mostly discuss range queries but our solutions can be easily adapted to support nearest-neighbor queries.

We briefly review the basics of answering queries with spatial conditions, and answering approximate string queries. Queries with a spatial condition are typically supported by a tree-based index such as an R*-tree, KD-tree, Quad-tree, etc. We assume a tree-based spatial index as a basis for our work, and we often use the R*-tree as a representative tree-based spatial access method. To support the approximate keyword condition, we use an index for approximate string search. Most trie-based indexes are specific to edit distance and its variants, and do not support other similarity measures such as Jaccard. An inverted-index approach [11] supports a family of similarity metrics, such as edit distance, Jaccard, etc. The two methods differ in their performance characteristics, whose specifics are independent of our proposed framework. We focus on the inverted-index approach, though we can as well choose a trie-based approach.

3. BASIC INDEX AND SEARCH

In this section, we introduce a basic index structure and the corresponding search algorithm for answering location-based approximate-keyword (LBAK) queries.

3.1 The LBAK-Tree

The main idea of the basic index (called "LBAK-tree") is to augment a tree-based spatial index with capabilities for approximate string search and keyword search. We use approximate string search to identify for each query keyword those strings that are similar. Once we have identified similar keywords, we use the keyword-search capability to prune search paths. For example, Figure 1 shows an LBAK-tree, e.g., an R*-tree that has been enhanced to support spatial approximate-keyword queries. To support keyword search we choose some nodes to store the union of keywords contained in objects of their subtree. To support approximate string search, we select nodes in the tree to build approximate string indexes (called "approximate indexes" hereafter) on their stored keywords. In this example and later discussions, we use a gram-based inverted index to perform approximate string search, but our solutions naturally extend to other types of approximate string indexes.

Fixed-level solution: A simple way to choose nodes to place approximate indexes on is the following. We choose one level L for which we construct approximate indexes at each node. The challenge is to choose a level L that provides good query performance. Notice that at each tree node, its stored keywords are the *union* of keywords of the objects in its subtree. If multiple objects have the same keyword, this keyword is stored only once in their common ancestors. There is a trade off between the average query time and the size of the approximate indexes. As we move L down the tree, the total number of keywords on which we need to build approximate indexes increases. Thus the total size of the approximate indexes will increase. Meanwhile, the performance of finding similar keywords from an approximate index is related to the size of the index. Typically the smaller the index is, the faster its lookups are. On the other hand, doing many approximate-keyword lookups on small indexes may increase the total running time as well.

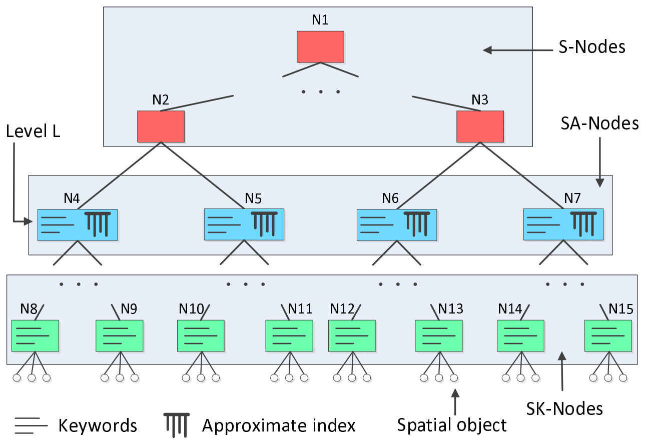


Figure 1: The LBAK-tree with approximate indexes at one level, and nodes enhanced with keywords.

3.2 Search Algorithm

Here, we discuss the search procedure for answering spatial approximate-keyword queries. We classify the LBAK-tree nodes into three categories:

- *S-Nodes* do not store any textual information such as keywords or approximate indexes, and can only be used for pruning based on the spatial condition.
- *SA-Nodes* store the union of keywords of their subtree, and an approximate index on those keywords. We use SA-Nodes to find similar keywords, and to prune subtrees with the spatial and approximate keyword conditions.
- *SK-Nodes* store the union of keywords of their subtree, and prune with the spatial condition and its keywords. Note that we must have previously identified the relevant similar keywords once we reach an SK-Node.

Algorithm outline: Let Q be a query with a spatial condition Q_s and an approximate-keyword condition Q_t with k keywords. The algorithm traverses the tree top-down and performs the following actions at a tree node depending on the node type. At an S-Node, the algorithm only relies on the spatial information of the node to decide which children to traverse. Once the algorithm reaches an SA-Node, it uses the node's approximate index to find similar keywords. For each keyword w_i in Q_t, the algorithm maintains a set of keywords C_i that are similar to w_i according to its similarity threshold δ_i. Assuming we use the AND-semantics, a node is pruned if one of the query's C_i sets is empty. Otherwise, we propagate the list $C = C_1, \ldots, C_k$ downward and use it for further pruning. In particular, at each SK-Node n, for each keyword w_i in Q_t, the algorithm intersects its similar keywords C_i propagated from the parent with the stored keywords of n. The node n can be pruned if one of the similar keyword-sets C_i has an empty intersection with the node's keywords. Otherwise, the algorithm passes those intersection sets of similar keywords to n's children for further traversal. At a leaf node, the algorithm adds to the answer set all the node's objects that satisfy the condition Q_s and have a non-empty intersection between their keywords and each of the propagated similar-keyword sets from the query.

Pseudo-code: Let us examine the pseudo-code for LBAK-Search in Algorithm 1. Note that the algorithm invokes several helper procedures, e.g. InitSimilarKeywordSets, ProcSNode, etc., defined in Algorithms 2, 3, 4 and 5. In later sections, we will override these helper procedures, but the procedure in Algorithm 1 remains the same.

The input of Algorithm 1 is a query $Q = \langle Q_s, Q_t \rangle$ and an LBAK-tree root r. We initialize a list C of k similar-keyword sets (line 2), where k is the number of keywords in Q_t. We maintain a stack S to traverse the tree. Initially, we push the root r and the list C to the stack (line 4). We start traversing the tree by popping the pair (n, C) from the stack (line 6). If n is not a leaf node, then all n's children that satisfy the spatial condition will be investigated (lines 7-9). Depending on the type of the node we invoke a helper procedure to process it (lines 10-18):

For an S-Node (lines 11-12), we only rely on the spatial condition to do pruning, and push the pair (n_i, C) to the stack S (within Algorithm 3). For an SA-Node (lines 13-14), we use its approximate index to find similar keywords for each query keyword as shown in Algorithm 4. We call GetSimKwds (w_i, δ_i) to get w_i's similar keywords, for $i = 1, \ldots, k$, and store them in w_i's corresponding similar-keyword set C_i. If at least one similar keyword is found for each keyword in Q_t, then the pair (n_i, C) is pushed to the stack S for future investigation. For an SK-Node (lines 15-16), we compute the intersection G_i between n_i's keywords and the similar-keyword set C_i. If all the intersection sets are not empty, then the pair (n_i, G) is pushed to S to be examined later (Algorithm 5). Finally, when we reach a leaf node, we add its objects that satisfy the two conditions Q_t and Q_s to the results (lines 20-26).

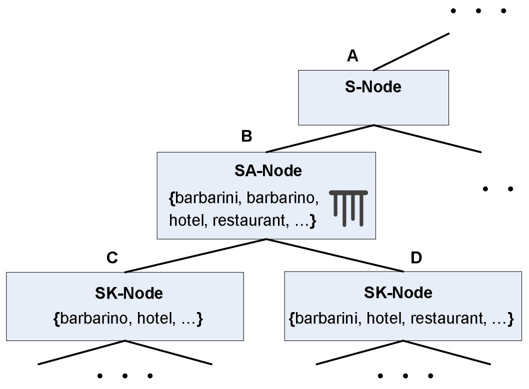

A
S-Node
B
SA-Node
{barbarini, barbarino, hotel, restaurant, ...}
C
SK-Node
{barbarino, hotel, ...}
D
SK-Node
{barbarini, hotel, restaurant, ...}

Figure 2: Exemplary portion of an LBAK-tree with approximate indexes at a fixed level.

Example: Figure 2 shows a portion of an LBAK-tree.

Algorithm 1: LBAKSearch

```
Input  : A query $Q = \langle Q_s, Q_t \rangle$, with $Q_t$ having $k$
         pairs $\{\langle w_1, \delta_1 \rangle, \ldots, \langle w_k, \delta_k \rangle\}$ associating a
         keyword $w_i$ with its similarity threshold $\delta_i$;
         An LBAK-tree root $r$;
Output: A set $R$ of all objects satisfying $Q_s$ and $Q_t$;
1  Result set $R \leftarrow \emptyset$;
2  $C \leftarrow$ InitSimilarKeywordSets($r$, $Q_t$);
3  Initialize an empty stack S;
4  $S$.push $(r, C)$;
5  while $S \neq \emptyset$ do
6      $(n, C) \leftarrow S$.pop();
7      if $n$ is not a leaf node then
8          foreach child $n_i$ of $n$ do
9              if $n_i$ does not satisfy $Q_s$ then continue;
10             switch $n_i$.type do
11                 case S-Node:
12                     $S \leftarrow$ ProcSNode($n_i$, $Q_t$, $C$, $S$);
13                 case SA-Node:
14                     $S \leftarrow$ ProcSANode($n_i$, $Q_t$, $C$, $S$);
15                 case SK-Node:
16                     $S \leftarrow$ ProcSKNode($n_i$, $Q_t$, $C$, $S$);
17                 endsw
18             endsw
19         end
20     else // leaf node
21         foreach object $o_i$ of $n$ do
22             if $o_i$ satisfies $Q_s$ and $Q_t$ then
23                 $R$.add($o_i$);
24             end
25         end
26     end
27 end
28 return $R$
```

We wish to find objects in `New York` with keywords similar to `barbarene` and `resturant`, expressed as: $Q = \langle \{\texttt{New York}\}; \{\langle \texttt{barbarene}, 2\rangle, \langle \texttt{resturant}, 2\rangle\}\rangle$. Notice that the query keywords have typos. We use edit distance as the similarity measure, and 2 as the similarity threshold for both keywords. Let us assume that nodes A, B, C, and D satisfy the spatial condition `New York`. Throughout the traversal of the tree, we always check the spatial condition. We focus on how to utilize the approximate indexes and stored keywords to prune irrelevant subtrees. At the S-Node A, we only rely on the spatial condition for pruning. When we reach the SA-Node B, we probe its approximate index to find keywords similar to `barbarene` and `resturant` according to the edit-distance threshold of 2. We can find two keywords similar to `barbarene` (namely, `barbarini` and `barbarino`), and one keyword similar to `resturant` (namely, `restaurant`). Once we visit the SK-Nodes C and D, we intersect their stored keywords with {`barbarini`, `barbarino`} and {`restaurant`}, respectively. Clearly, we can prune node C because it does not have the keyword `restaurant`. Since node D has the keywords `barbarini` and `restaurant`, we traverse its children. We repeat the process until we find the answers.

Algorithm 2: InitSimilarKeywordSets

```
Input  : Root $r$ of an LBAK-tree;
         $Q_t = \{\langle w_1, \delta_1 \rangle, \ldots, \langle w_k, \delta_k \rangle\}$
Output: A list of similar-keyword sets
         $C = C_1, \ldots, C_k$;
1  $C \leftarrow \{\emptyset, \emptyset, \ldots, \emptyset\}$ // $k$ empty sets
2  return $C$
```

Algorithm 3: ProcSNode

```
Input  : S-Node $n$;
         $Q_t = \{\langle w_1, \delta_1 \rangle, \ldots, \langle w_k, \delta_k \rangle\}$;
         Similar-keyword sets $C = C_1, \ldots, C_k$;
         Stack $S$;
Output: Stack $S$;
1  $S$.push $(n, C)$;
2  return $S$
```

Algorithm 4: ProcSANode

```
Input  : SA-Node $n$;
         $Q_t = \{\langle w_1, \delta_1 \rangle, \ldots, \langle w_k, \delta_k \rangle\}$;
         Similar-keyword sets $C = C_1, \ldots, C_k$;
         Stack $S$;
Output: Stack $S$;
1  for $i$=1 to $k$ do
2      $C_i \leftarrow n$.GetSimKwds($w_i$, $\delta_i$);
3  end
4  if all $C_i$'s $\neq \emptyset$ then
5      $S$.push $(n, C)$;
6  end
7  return $S$
```

4. PLACING APPROXIMATE INDEXES AT VARIABLE LEVELS

In this section we study how to improve the solution described in Section 3 based on the following observations. First, our experiments showed that around 90% of the query time is spent on approximate-index lookups. Therefore, optimizing the placement of approximate indexes in the tree can greatly improve the average query time.

Second, the fixed level L is chosen without considering the local spatial distribution of the objects within each node. In

Algorithm 5: ProcSKNode

```
Input  : SK-Node $n$;
         $Q_t = \{\langle w_1, \delta_1 \rangle, \ldots, \langle w_k, \delta_k \rangle\}$;
         Similar-keyword sets $C = C_1, \ldots, C_k$;
         Stack $S$;
Output: Stack $S$;
1  for $i$=1 to $k$ do
2      $G_i \leftarrow n$.keywords $\cap C_i$;
3  end
4  if all $G_i$'s $\neq \emptyset$ then
5      $G \leftarrow G_1, G_2, \ldots, G_k$;
6      $S$.push $(n, G)$;
7  end
8  return $S$
```

many datasets, the spatial distribution of objects is skewed, and nodes at the same level can greatly vary in size. For instance, consider a sparse node n_1 in an R*-tree, e.g., a desert area in Nevada, where very likely a query overlaps with only few of n_1's children. When traversing the tree through n_1, it is better to rely only on the very selective spatial condition for pruning without considering its textual information. Thus, we prefer to build the approximate indexes at the descendants of n_1, where the local pruning power of the spatial condition becomes weaker. Our hope is that a query will only probe a few, small approximate indexes below n_1, if any. On the other hand, consider a dense node n_2, e.g., one representing New York city, where a query region is likely to overlap with many of n_2's children. If we build approximate indexes for n_2's children, a query will likely need to probe all of them, which would be more expensive than probing one big approximate index at n_2. Therefore, we would rather build a single approximate index at n_2 to avoid the cost of many approximate-index lookups.

Our new method presented in this section allows the approximate indexes to appear at different levels in the tree as shown in Figure 3. The new method reduces both the overall space cost of the index and the average query time.

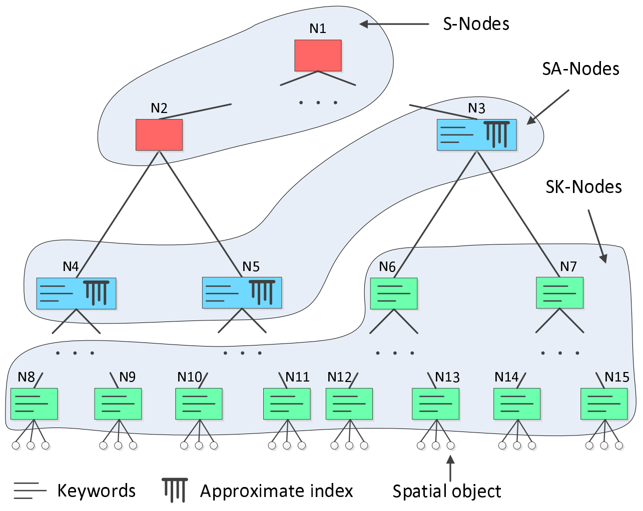


Figure 3: The LBAK-tree utilizing the local spatial distribution of objects.

4.1 Selecting Nodes for Approximate Indexes

Our problem is finding the optimal set of nodes that should have approximate indexes. It can be formulated as the following optimization problem: "Given an R*-tree and a space budget, choose nodes from the tree to store approximate indexes, such that the average query time of a given workload is minimized." We can show that this problem is NP-hard by a reduction from the Knapsack problem to this problem, where each approximate index has a value (the average query time improvement that we obtain by building the index) and a weight (the space cost of the index).

Algorithm outline: We develop a greedy algorithm, called SelectSANodes, that traverses the tree top-down and tries to push approximate indexes down the most promising paths. The algorithm maintains a priority queue of nodes to be visited. The priority of a node n is the benefit of building multiple approximate indexes at its children as compared to building a single index at n. A detailed explanation of how to compute this benefit is presented in Section 4.2. For each visited node n, if the benefit of building multiple approximate indexes at n's children is negative, then the algorithm selects n to be an SA-Node, and it will not traverse its children. If the algorithm reaches a leaf node, it immediately selects the leaf to be an SA-Node. The algorithm terminates when the space budget is exhausted or there is no more benefit to pushing approximate indexes down the tree.

Pseudo-code: We now walk through the pseudo-code of SelectSANodes as shown in Algorithm 6. We use W_n to denote the set of stored keywords at node n. The input of the algorithm is an R*-tree root r and a space budget S. We assume that the space budget is large enough to at least build an approximate index on the root's keywords W_r. We maintain a priority queue H of nodes to visit, ordering the nodes descending by their benefit. Initially, we call GetBenefit (r) to obtain the benefit of storing the approximate indexes at r's children (line 3), and push r with its benefit to the queue (line 4). The algorithm starts traversing the tree by popping the pair with the highest benefit to (n, B) (line 6). Next, we call GetSpaceCost (W_{n_i}) to compute the cost of building multiple approximate indexes at n's children (lines 7-10). We choose to build a single approximate index at n, and adjust the space budget S accordingly if one of the following is true: n is a leaf node, there is not enough space to build multiple approximate indexes at n's children, or there is no benefit of building multiple approximate indexes at the children (lines 11-13). Otherwise, we push n's children to the queue for further evaluation (lines 14-19).

Algorithm 6: SelectSANodes

```
Input  : An R*-tree root $r$;
         A space budget $S$;
Output: A set of nodes $R$ to build approximate
         indexes on;
1  Result set $R \leftarrow \emptyset$;
2  Priority Queue $H \leftarrow \emptyset$;
3  $B \leftarrow$ GetBenefit($r$);
4  $H$.push (r, B);
5  while $H \neq \emptyset$ do
6      $(n, B) \leftarrow H$.pop();
7      $S_c \leftarrow \emptyset$; // space cost of children
8      foreach child $n_i$ of $n$ do
9          $S_c \leftarrow S_c +$ GetSpaceCost($W_{n_i}$);
10     end
11     if $n$ is a leaf node or $S <= S_c$ or $B <= 0$ then
12         $R$.add($n$);
13         $S \leftarrow S-$ GetSpaceCost($W_n$);
14     else
15         foreach child $n_i$ of $n$ do
16             $B_i \leftarrow$ GetBenefit($n_i$);
17             $H$.push $(n_i, B_i)$;
18         end
19     end
20 end
21 return $R$
```

The search algorithm for answering queries based on an index built with SelectSANodes is the same as the one for the fixed-level solution, i.e., Algorithm 1 with the helper Algorithms 2, 3, 4, and 5.

4.2 Estimating Benefits

In this section, we develop a cost model for algorithm SelectSANodes to estimate the benefit of pushing approximate indexes down the tree based on the expected size overhead and lookup-time improvement. Recall that we use the benefits to order the nodes in the priority queue.

Benefit of a node: Given a node n and its m children $n_1, \ldots, n_m$, the benefit of n, denoted as $b(n)$, is determined by the ratio of the lookup-time difference of a single approximate index at n and multiple approximate indexes at n's children, to their size difference. The following equation expresses this idea:

$$b(n) = \frac{pTime(n) - cTime(n)}{|pSpace(n) - cSpace(n)|}, \quad (1)$$

where "pTime" is the average query time of probing an approximate index at the parent, "cTime" denotes this time if the indexes were built at the children, and "pSpace" and "cSpace" are the corresponding space costs of the indexes.

Let W_n denote the set of stored keywords at node n, $s(W_n)$ denote the size of an approximate index on keywords W_n, and $t(W_n)$ denote the average lookup time for that approximate index. We define the benefit of node n as follows:

$$b(n) = \frac{p(n) * t(W_n) - \sum_{i=1}^{m} p(n_i) * t(W_{n_i})}{|s(W_n) - \sum_{i=1}^{m} s(W_{n_i})|}. \quad (2)$$

We weight the lookup time for a node n by $p(n)$, which denotes the probability of n satisfying the spatial condition of a query in a workload.

Space cost of an approximate index: Since the main space cost of a gram-based approximate index is the inverted lists, we focus on estimating their size.

Suppose we use q-grams in the approximate indexes for a positive integer q. Each keyword w to be inserted into the approximate index yields $|w| - q + 1$ q-grams. Let $G(w)$ be the bag of grams generated for w. For every gram in $G(w)$, we insert one element into its corresponding inverted list. Let σ be the size of each inverted-list element. The keyword w contributes at most $(|w| - q + 1) * \sigma$ bytes to the size of the inverted lists. Note that this estimate is an upper bound on w's size contribution because we remove duplicate string ids from inverted lists. Thus, we estimate the size of an approximate index on a set of keywords W as:

$$s(W) = |W| * (\lambda - q + 1) * \sigma, \quad (3)$$

where λ is the average keyword length of a particular dataset.

Lookup time of an approximate index: The lookup time of an approximate index is a function of its size. We have experimentally determined that the cost of querying an approximate index is linear in the number of keywords. Table 1 shows the experimental data underlying this conclusion. We built four approximate indexes of different sizes (first column), and ran a workload of queries against them to determine the average lookup time (second column). In the third column we computed the slope of the line going through the point represented by the first row and the other corresponding point. Since the slope is roughly constant for different index sizes, we conclude that the average approximate-index lookup time grows linearly with its size. Thus, we can estimate the average lookup time of an approximate index on W keywords as:

$$t(W) = \beta * |W| + \alpha, \quad (4)$$

the slope β and the intercept α are implementation dependent and can be determined experimentally, as we did.

Size	Time	Slope
1	0.02	-
10000	0.207	0.000019
1M	22.253	0.000022
10M	210.152	0.000021

Table 1: Experimental data showing linear growth in lookup-time with size of an approximate index.

5. EXPLOITING FREQUENCY DISTRIBUTION OF KEYWORDS

Objects in a spatial dataset can share the same keyword. For example, many objects can have the keyword `hotel` in their textual descriptions. Moreover, very often some keywords appear more frequently than others, and their frequency distribution is skewed. For instance, in a dataset about business listings, a keyword `restaurant` is more likely to appear frequently in objects than `consulate`. In Section 4, we presented a cost-based solution that selects a set of nodes to store the approximate indexes. In this section, we further improve our cost-based index-construction procedure by exploiting the skewed distribution of keywords. The following enhancements can further reduce both the space cost of the approximate indexes and the average query time.

5.1 Index Construction

Intuitively, we want to reduce the number of keywords in the approximate indexes. We achieve this by removing frequent keywords from sibling nodes, and placing them in their common parent instead. As a consequence, approximate indexes on frequent keywords can now appear in any S-Node above an SA-Node. To further clarify, as shown in Figure 4, SA-Nodes contain approximate indexes on infrequent keywords, while some S-Nodes above an SA-Node hold approximate indexes on frequent keywords.

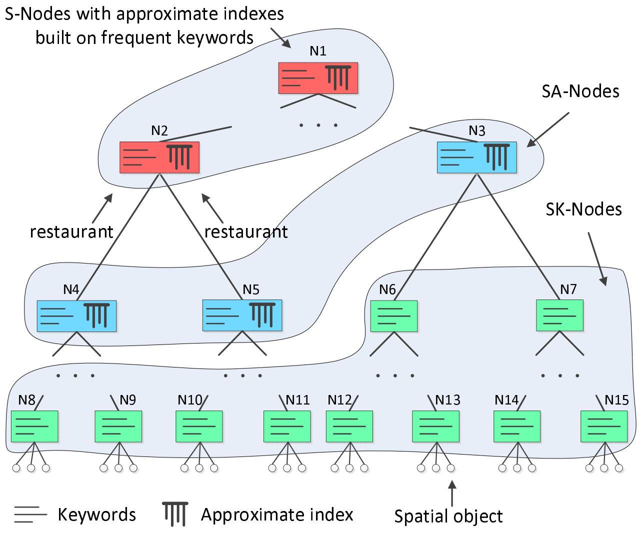


Figure 4: Improved LBAK-tree exploiting the skewed frequency distribution of keywords.

A keyword is considered frequent in a node n if the fraction of n's children having that keyword exceeds a certain threshold, denoted by ω. For example, if $\omega = 0.9$, then a keyword is frequent if it appears in at least 90% of n's children. A small ω decreases the space cost of the approximate indexes. On the other hand, the average query time may increase because we could visit false-positive nodes, since not all of n's children actually contain the frequent keywords. Those false-positive paths will be pruned at SK-Nodes.

Algorithm outline: To discover the frequent keywords in the tree, we maintain for each node n two sets of keywords: a set of infrequent keywords W_n, and a set of frequent keywords F_n. We identify the frequent and infrequent keywords of the node n by examining its children. We say a child n_i has a particular keyword if the keyword occurs in W_{n_i} or F_{n_i}. If a keyword w is frequent at node n, then we remove w from the appropriate keyword sets in all of n's children. In this way, we ensure that popular keywords in a subtree only appear in the root of that subtree. The propagation of frequent and infrequent keywords is performed bottom-up until the keyword sets of all nodes have been filled. The next step is to choose nodes to build approximate indexes on. We use a procedure similar to SelectSANodes from Section 4 but with a modified benefit function that distinguishes the frequent and infrequent keywords. Since the index-construction procedure requires only minor modifications, we omit its pseudo-code.

Benefit of a node: The new benefit of a node n is determined by the space and time cost of building an approximate index on $W_n \cup F_n$, versus building multiple indexes at n's children excluding the frequent keywords at n. We use the following variations for pTime, cTime, pSpace, and cSpace in the benefit function of Equation 1:

$$\begin{aligned}
&pTime(n) = p(n) * t(W_n \cup F_n)\\
&cTime(n) = p(n) * t(F_n) + \sum_{i=1}^{m} p(n_i) * t((W_{n_i} \cup F_{n_i}) - F_n)\\
&pSpace(n) = s(W_n \cup F_n)\\
&cSpace(n) = s(F_n) + \sum_{i=1}^{m} s((W_{n_i} \cup F_{n_i}) - F_n)
\end{aligned} \tag{5}$$

As before, we use $p(n)$ to denote the probability of n satisfying the spatial condition of any query in a workload.

5.2 Search Algorithm

Algorithm 7: InitSimilarKeywordSets

```
    Input/Output: Same as Algorithm 2
1 $C \leftarrow \{\emptyset, \emptyset, \ldots, \emptyset\}$ // k empty sets
2 for $i$=1 to $k$ do
3     $C[i] \leftarrow r$.GetSimKwds($w_i$,$\delta_i$);
4 end
5 return $C$
```

Answering approximate-keyword queries on the improved index follows the common search Algorithm 1, with different implementations for the helper procedures InitSimilarKeywordSets, ProcSNode and ProcSANode. We first probe the approximate index (if any) at the root node to get the similar query keywords that are frequent at the root (Algorithm 7). During the tree traversal, when we encounter an S-Node with an approximate index (on frequent keywords), we probe it for similar keywords (Algorithm 8). Note that at such an S-Node, we cannot prune subtrees based the textual condition, because we cannot guarantee to have gathered all similar keywords yet (we have only gathered those similar keywords that are frequent). At an SK-Node we probe its approximate index, and possibly prune its subtree based on the textual information (Algorithm 9).

Algorithm 8: ProcSNode

```
    Input/Output: Same as Algorithm 3
1 for $i$=1 to $k$ do
2     $G_i \leftarrow C_i \cup n_i$.GetSimKwds($w_i$)
3 end
4 $G \leftarrow G_1, G_2, \ldots, G_k$;
5 $S$.push ($n_i, G$);
6 return $S$
```

Algorithm 9: ProcSANode

```
    Input/Output: Same as Algorithm 4
1 for $i$=1 to $k$ do
2     $G_i \leftarrow C_i \cup n_i$.GetSimKwds($w_i$,$\delta_i$)
3 end
4 if all $G_i$'s $\neq \emptyset$ then
5     $G \leftarrow G_1, G_2, \ldots, G_k$;
6     $S$.push ($n_i, G$);
7 end
8 return $S$
```

5.3 Incremental Maintenance of Indexes

We discuss how to incrementally maintain the proposed indexes in the presence of insertions and deletions. Insertion of new objects into an LBAK-tree proceeds as follows. We first insert the new object into a leaf according to the standard R*-tree insertion procedure (assuming no node-splits for now). We then propagate the keywords of the new object bottom-up. At an SK-Node we add the new keywords to its stored set of keywords. At an SA-Node we add the keywords to its approximate index. For the approach exploiting keyword frequencies, we must pay attention to add the new keywords to the appropriate keyword set, i.e., the one with frequent or infrequent keywords. Additionally, at an S-Node we check its children for new frequent keywords, and add them to its approximate index. We deal with R*-tree node-splits as follows. For the two new nodes generated by the split, we recompute their stored set of keywords (frequent and infrequent) by examining their children. If the nodes are SA-nodes, then we delete their approximate indexes, and give them a special "split" marker. After we have propagated all new keywords up to the root, we re-traverse the tree and rebuild approximate indexes at those nodes with special markers (and remove the markers). Note that the R*-tree may cause re-insertion of objects, causing multiple splits in different branches. Therefore, the second re-traversal of the tree is necessary if a split occurred. Deletions can be dealt with in a similar fashion. Before deleting a keyword at a particular node, we must examine all its children to ensure *none* of them have that keyword (or if the deletion causes a frequent keyword to become infrequent).

If many updates significantly change the distribution of keywords, we can reevaluate for each SA-Node whether pushing their approximate index up or down the tree would yield a benefit (according to the appropriate benefit function). We can use a similar procedure to deal with changing workloads, but must additionally modify the intersection probabilities ($p(n)$) in the benefit function to reflect the new workload.

6. EXPERIMENTS

In this section, we present our experimental results on the proposed techniques. We evaluated the following variations: the fixed-level (FL) approach from Section 3 (e.g., "FL-0" means the approximate indexes are at the root level), the variable-level approach (VL) from Section 4, and the variable-level approach exploiting keyword-frequencies (VLF) from Section 5. We used the Flamingo Package [3] as our approximate string-search solution. We also compare our solutions with the MHR-tree [14]. We conducted all experiments on a machine with a four-core Intel Xeon E5520 2.26Ghz processor and 12GB of RAM, running a Ubuntu operating system. We implemented all algorithms (including the R*-tree) in C++ and compiled them with GCC using the "$-$O3" flag. We stored the index structures in main memory in order to achieve the goal of a high query throughput, as required by many online search engines. If not stated otherwise, we use a keyword-frequency threshold of $\omega = 1$ for algorithm VLF, and a branching factor of 40 for the R*-tree.

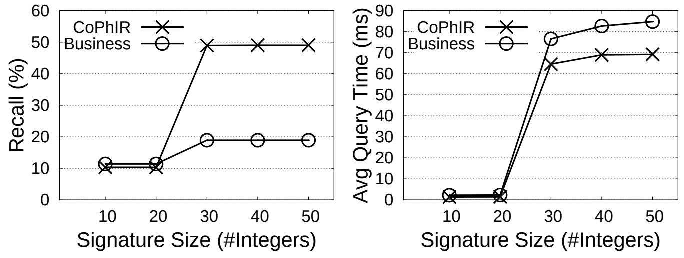
Datasets: We used two real, large datasets. The first one was a multimedia metadata collection extracted from Flickr pages, called "CoPhIR Test Collection" [4]. We processed the dataset to extract the photos taken in the U.S. based on their latitude and longitude values. Moreover, we used the keywords in the title, description, and tags of a photo as its textual attribute. The average number of keywords per object was 13.5. The final dataset had about 3.75 million objects. The total raw data size was about 500MB. We refer to this dataset as CoPhIR. The second dataset contained 20.4 million business listings in the U.S., obtained from Florida International University.[2] Each business listing had a longitude and latitude value, and a descriptive name consisting of three keywords on average. The total raw data size was about 4GB. We refer to this dataset as Business.

Queries: We generated a workload of 10,000 queries for each dataset as follows. We randomly selected objects from the dataset and used their coordinates as the query-window center. We then created a 30km-by-30km query window around that center, to reflect a spatial condition. For the approximate keyword condition, we randomly chose two of the keywords of the randomly chosen object. For most of the experiments we used a normalized edit-distance function and a similarity threshold of 0.8.

6.1 Comparison with MHR-Tree

We first compared our LBAK-tree constructed using the VLF algorithm with the MHR-tree [14]. We used an edit distance threshold of 2 for both approaches. The main difference between these two approaches is that the MHR-tree uses a probabilistic signature scheme to represent textual information in tree nodes, whereas the LBAK-tree uses approximate indexes and keywords for that purpose. Since the min-wise signatures in the MHR-tree are probabilistic in nature, this approach could miss some answers. On the other hand, the size overhead of these signatures is very small. Figure 5 shows the main issues with MHR-tree. First, the recall of MHR-tree shown in Figure 5(a) were constantly below 50%, which may not be acceptable for many applications. Second, as we increased the signature size in order to achieve a higher recall, the query time also increased (Figure 5(b)). The reason is that the pruning power of the min-wise signatures is limited, and the approximate keyword condition is validated at the leaf level, leading to many edit-distance computations as the recall increases.

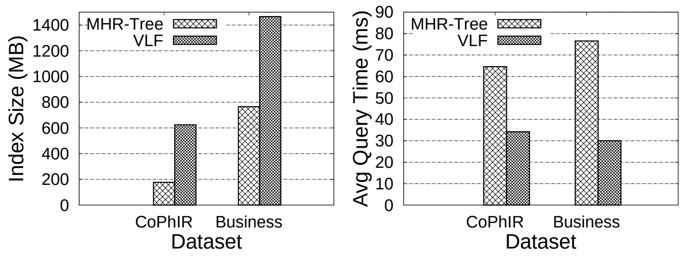
[2]http://n0.cs.fiu.edu/ihmc.com.forms.html



(a) Recall (b) Query Time

Figure 5: Recall and query time of MHR-tree with increasing signature size.

In Figure 6 we compare VLF with MHR-tree using 30 signatures for each tree node. Clearly, the merit of MHR-tree lies in a comparably small index size, due to the compact signatures. However, we see that VLF significantly outperformed MHR-tree in terms of query time. Note that as shown in Figure 5(b), the query time of MHR-tree will likely increase if we give it more space.

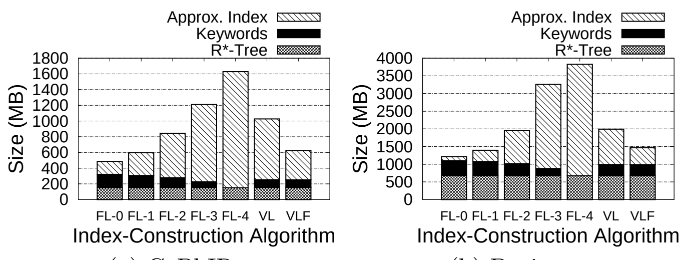


(a) Index Size (b) Query Time

Figure 6: Comparison of VLF with MHR-tree using 30 signatures for each node.

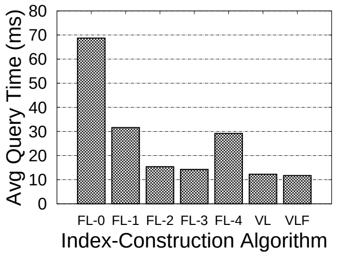
6.2 Index Size and Query Time

Figure 7 shows the sizes of the individual index components for various construction algorithms. Figure 8 shows the corresponding query times. For the algorithms VL and VLF, we report the minimum index size required to achieve the best query performance.

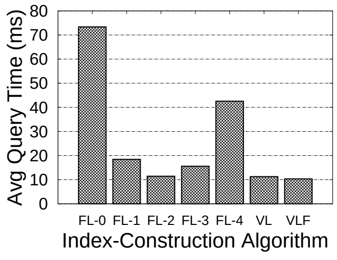


(a) CoPhIR (b) Business

Figure 7: Sizes of index components.



(a) CoPhIR



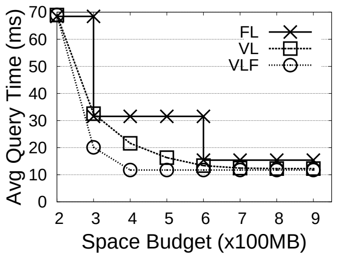
(b) Business

Figure 8: Query time by index-construction method.

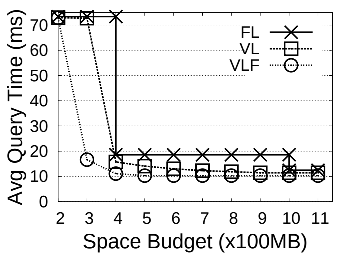
The fixed-level solutions FL-0 to FL-4 show a clear size-trend in Figure 7: as we pushed the approximate indexes down the tree, their space requirement increased because of redundant keywords in adjacent nodes. On the other hand, the query times decreased because we probed few, smaller indexes rather than one bigger index. But the query time eventually increased when pushing the indexes further down (e.g., FL-3 in Figure 8(b)), again, because of keyword redundancies. The space overhead for the approximate indexes for algorithm VL was large compared to the R*-tree and keywords, but the query time was good. Compared to VL, algorithm VLF pushed frequent keywords up the tree to cut the space by half without sacrificing query performance.

6.3 Space Budget

In Figure 9 we show the effect on query performance when giving our index-construction methods FL, VL, and VLF a space budget for building approximate indexes.



(a) CoPhIR



(b) Business

Figure 9: Query time with increasing space budget for approximate indexes.

The fixed-level solution, FL, exhibited "jumps" when given enough space to push down the approximate indexes one more level, improving the query time. Note that at even higher space budgets not shown in the figure, the query time of FL will eventually increase, due to the cost of probing many, smaller approximate indexes at a lower level. VL's and VLF's curves are smoother than FL's because they have more flexibility in placing the approximate indexes. VL's curve meets FL's curve at some points because their performance is limited by redundant keywords residing in many approximate indexes. This observation is supported by VLF outperforming both LF and VL at the points where LF and VL meet. In summary, VLF offers good query performance at significantly less space than the other two methods.

6.4 Scalability

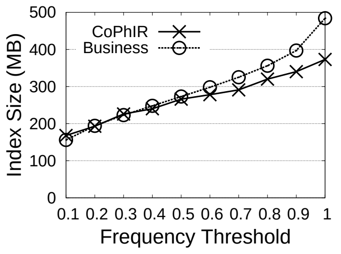
We varied the number of objects indexed by different LBAK-tree variants. Figures 10(a) and 11(a) show the total index sizes. For algorithms VL and VLF, we report the minimum index size required to achieve the best query performance. Figures 10(b) and 11(b) show the corresponding best query times. To emphasize the merit of VLF, we created Figures 10(c) and 11(c) as follows. We determined the minimum index size for VLF to achieve the best query time, and used that size as a space budget for FL and VL. As a result, FL's level could vary from point to point.

For both datasets, we observe a linear trend for the index size and query time (Figures 10 and 11(a,b)). The fixed-level approaches show a space-time tradeoff with the level. As we pushed the approximate indexes down the tree the index size increased because the number of duplicate occurrences of keywords in approximate indexes increased. However, the query time improved until FL-2 for 11(b) and FL-3 for 10(b), but then sharply degraded at lower levels. The reason is that the cost of probing multiple smaller approximate indexes became higher than probing a few larger ones. The effect of duplicate keywords can also be observed by comparing VL and VLF. VLF consistently performed better both in space and time, because it effectively minimized the redundancy in keywords. Notice that space and time increased more rapidly on the CoPhIR dataset than on the Business dataset. The main reason is that the objects in the CoPhIR dataset had, on average, a higher number of keywords in their description. The longer textual description contributed to the index size, and increased the chance of intersecting with query keywords. In comparison, the Business dataset was sparse in the textual dimension, making queries highly selective and rendering their performance insensitive to the number of indexed objects.

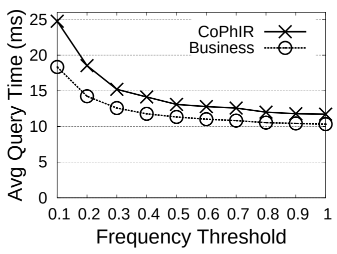
Finally, let us examine Figures 10(c) and 11(c). Here, the difference between VL and VLF is clearer due to the smaller scale of the Y-axis (as compared to the other figures). We see that when given the same amount of memory, VLF clearly outperformed the other methods, though they could achieve a comparable performance with more memory (Figure 9).

6.5 Keyword-Frequency Threshold

Algorithm VLF uses a threshold ω to decide whether a keyword is frequent or not (Section 5). Intuitively, a threshold of $\omega = 0$ (i.e., every keyword is considered to be frequent) would produce an LBAK-tree with one approximate index at the root. On the other hand, a threshold of $\omega > 1$ (no keyword is frequent) would produce an LBAK-tree similar to the one generated by VL. We ran experiments with varying ω values, the results of which are shown in Figure 12.



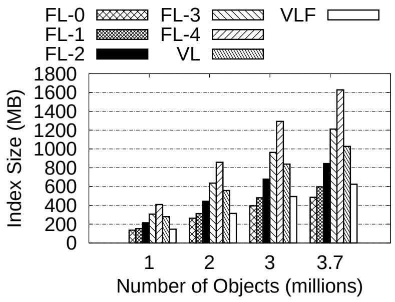
(a) Approximate-Index Size



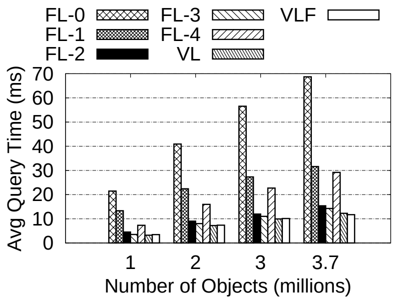
(b) Query Time

Figure 12: Effect of keyword-frequency threshold.

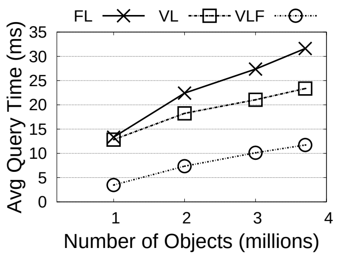
We observe a clear space-time tradeoff with the keyword-frequency threshold. As we increased the threshold, we pushed more keywords to lower levels in the tree, causing space overhead due to infrequent keywords being duplicated



(a) Total Index Size.

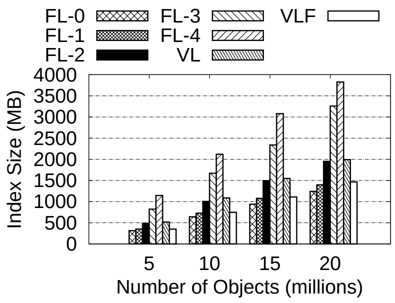


(b) Query Time.

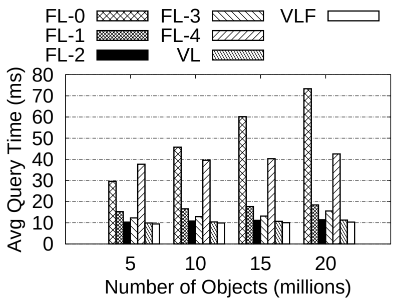


(c) Query Time vs. VLF.

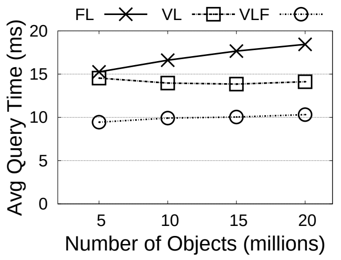
Figure 10: Index size and query time with varying numbers of indexed objects on CoPhIR.



(a) Total Index Size.



(b) Query Time.



(c) Query Time vs. VLF.

Figure 11: Index size and query time with varying number of indexed objects on Business.

at multiple nodes. On the other hand, increasing the threshold decreased the query time, because of the following two effects: (1) we traversed fewer false-positive branches that did not actually have a keyword deemed frequent at its parent, and (2) the total cost of probing a few smaller indexes at a node's children could be less than probing one big index at the node itself. The VLF algorithm will try to push indexes only down these beneficial paths.

7. CONCLUSION

In this paper we presented an index structure called LBAK-tree to answer location-based approximate-keyword queries. We showed how to combine approximate indexes efficiently with a tree-based spatial index. We developed a cost-based algorithm that selects tree nodes to store approximate indexes. Moreover, we improved the techniques to exploit the frequency distribution of keywords, further reducing the index size and query times. Finally, we conducted extensive experiments to show the efficiency of our techniques.

Acknowledgements: We thank Dr. Naphtali David Rishe for generously providing us with the business listings dataset. We also thank Shengyue Ji for his insightful discussions regarding this work.

8. REFERENCES

[1] S. Alsubaiee and C. Li. Fuzzy keyword search on spatial data (demo). In *DASFAA*, 2010.

[2] N. Beckmann, H. P. Begel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *SIGMOD*, 1990.

[3] A. Behm, R. Vernica, S. Ji, J. Lu, L. Jin, Y. Lu, and C. Li. UCI Flamingo Package 3.0, 2010.

[4] P. Bolettieri, A. Esuli, F. Falchi, C. Lucchese, R. Perego, T. Piccioli, and F. Rabitti. CoPhIR: a test collection for content-based image retrieval. *CoRR*, 2009.

[5] Y.-Y. Chen, T. Suel, and A. Markowetz. Efficient query processing in geographic web search engines. In *SIGMOD*, 2006.

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

[7] I. D. Felipe, V. Hristidis, and N. Rishe. Keyword search on spatial databases. In *ICDE*, 2008.

[8] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, 1984.

[9] M. Hadjieleftheriou, A. Chandel, N. Koudas, and D. Srivastava. Fast indexes and algorithms for set similarity selection queries. In *ICDE*, 2008.

[10] R. Hariharan, B. Hore, C. Li, and S. Mehrotra. Processing spatial-keyword (SK) queries in geographic information retrieval (GIR) systems. In *SSDBM*, 2007.

[11] C. Li, J. Lu, and Y. Lu. Efficient merging and filtering algorithms for approximate string searches. In *ICDE*, 2008.

[12] G. Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1), 2001.

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

[14] B. Yao, F. Li, M. Hadjieleftheriou, and K. Hou. Approximate string search in spatial databases. In *ICDE*, 2010.

[15] D. Zhang, B. C. Ooi, and A. K. H. Tung. Locating mapped resources in web 2.0. In *ICDE*, 2010.

[16] Y. Zhou, X. Xie, C. Wang, Y. Gong, and W.-Y. Ma. Hybrid index structures for location-based web search. In *CIKM*, 2005.