Location-Based Instant Search

Shengyue Ji and Chen Li*

University of California, Irvine

Abstract. Location-based keyword search has become an important part of our daily life. Such a query asks for records satisfying both a spatial condition and a keyword condition. State-of-the-art techniques extend a spatial tree structure by adding keyword information. In this paper we study location-based instant search, where a system searches based on a partial query a user has typed in. We first develop a new indexing technique, called filtering-effective hybrid index (FEH), that judiciously uses two types of keyword filters based on their selectiveness to do powerful pruning. Then, we develop indexing and search techniques that store prefix information on the FEH index and efficiently answer partial queries. Our experiments show a high efficiency and scalability of these techniques.

1 Introduction

Location-based services have become an important part of our daily life. We use online maps to search for local businesses such as stores and movie theaters; we use Yelp.com to search for restaurants; and we use Twitter to search for nearby tweets. On the back-end of these systems there are spatial records with keyword descriptions. For instance, Figure 1 shows an example data set that includes business listings in Manhattan, New York, such as museums, schools, and hospitals. Each record has a *Name* value (e.g., Metropolitan Museum of Art) and a *Location* value including the latitude and longitude of the entity (e.g., ⟨40.7786, -73.9629⟩). The entities are also shown as points on the map in the figure. These applications need to answer *location-based keyword queries*, a.k.a. spatial keyword search. A query includes a location, such as the point P or the area R in Figure 1. The query also includes keywords, and asks for answers that match the keywords and are close to the spatial location. Example queries include "finding movie theaters close to downtown New York" and "finding Japanese restaurants near the Disneyland in California."

Instant keyword search has become popular in recent years. It returns the search results based on partial query keywords as a user is typing. Users of an instant search system can browse the results during typing. In this paper, we study location-based instant search, a search paradigm that combines location-based keyword search with instant search. Figure 2 shows an interface of location-based instant search on our example data set. As the user types a query letter by letter, the system responds to the partial queries and returns the results to

* The author has financial interest in Bimaple Technology Inc., a company currently commercializing some of the techniques described in this publication.

ID	Name	Location
1	Trinity School	<40.7903, -73.9707>
2	Metropolitan Hospital Center	<40.7846, -73.9441>
3	Solomon R. Guggenheim Museum	<40.7831, -73.9596>
4	Brearly School	<40.7783, -73.9501>
5	Metropolitan Museum of Art	<40.7786, -73.9629>
6	American Museum of Natural History	<40.7791, -73.9730>
7	Manhattan Church of Christ	<40.7766, -73.9613>
8	Mt Sinai Hospital	<40.7901, -73.9538>
9	Cooper Hewitt Museum	<40.7844, -73.9580>
...	...	...

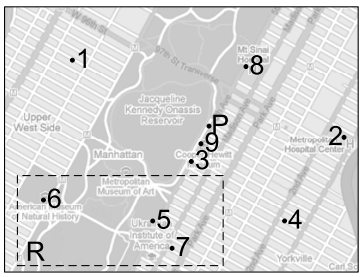


Fig. 1. Spatial keyword records of business listings. The map on the right shows the listings as well as a query area R and a query point P.

the user, listing them and plotting them on the map. When the system receives the partial query "`Metropolitan Mus`", it returns businesses that are nearest to Manhattan (represented as a point returned by the geo-coder), having the keyword `Metropolitan` and a keyword with `Mus` as a prefix.

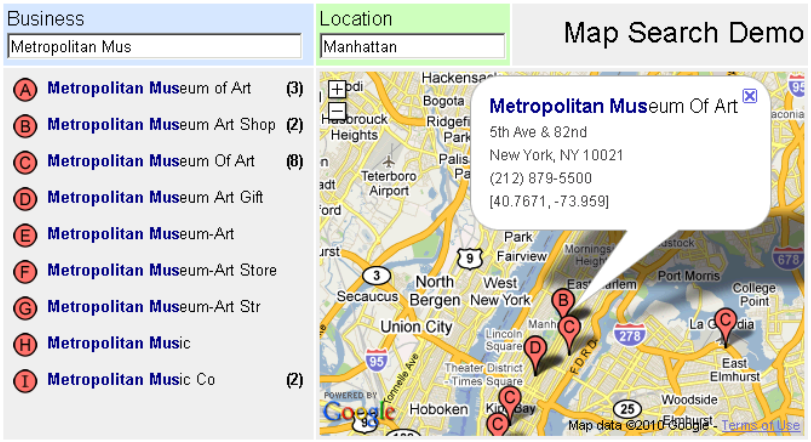


Fig. 2. Location-based instant search.

In these systems, it is critical to answer queries efficiently in order to serve a large amount of query traffic. For instance, for a popular map-search service provider, it is not uncommon for the server to receive thousands of queries per second. Such a high query throughput requires the search process to be able to answer each search very fast (within milliseconds). Instant search would further increase the server workload. Therefore, we focus on searching using in-memory index in this paper to achieve a high efficiency for instant search. There are recent studies on supporting location-based keyword queries. A common solution is extending a spatial tree structure such as R-tree or R*-tree by adding keyword information on the tree nodes. For example, Figure 3 in Section 2 shows such an index structure, in which each node has a set of keywords and their children with these keywords. We can use the keyword information to do efficient pruning during the traversal of the tree to answer a query, in addition to the pruning based on the spatial information in the tree.

In this paper we first present an index structure called "filtering-effective hybrid" (FEH) index. It judiciously uses two types of keyword filters in a node

of a spatial tree based on the selectiveness of each keyword. One filter, called *child filter*, maps keywords and their corresponding children nodes. Another filter, called "*object filter*", maps keywords to their corresponding records in the subtree of the node. During a traversal of the FEH index tree, the object filter at each node allows us to directly retrieve records for these keywords in the filter, thus bypassing those intermediate nodes in the subtree. Next we study how to answer location-based instant queries on spatial data, i.e., finding answers to a query as the user is typing the keywords character by character. We utilize existing indexing techniques and FEH to answer queries. We develop a technique to store prefix filters on spatial-tree nodes using a space-efficient representation. In addition, we develop a method to compress the representation in order to further reduce the index size.

We show that our techniques can be applied to efficiently support both range queries (where the user specifies a spatial area) and nearest-neighbor queries (where the user wants to find objects closest to a location). Our techniques can also reduce the index size. Such reduction can minimize the hardware cost. We have conducted a thorough experimental study to evaluate these techniques. The results show that, our techniques can support efficient instant search on large amounts of data. For instance, we are able to index a data set of 20 million records in memory on a commodity machine, and answer a location-based instant keyword search in microseconds.

The rest of the paper is organized as follows. In Section 2 we give the problem formulation of location-based instant search, and describe a state-of-the-art approach for answering location-based keyword queries. In Section 3, we present our FEH index. In Section 4, we study how to efficiently answer location-based instant queries, using existing index techniques and FEH. We report our experimental results in Section 5, and conclude in Section 6.

1.1 Related Work

Location-based keyword search received a lot of attention recently. Early studies utilize a keyword index (inverted lists) and a spatial index (such as R-tree [10] or R*-tree [4]) separately [22, 7, 18]. These proposed methods answer a query by using the keyword and spatial indexes separately. A main disadvantage of these approaches is that filtering on the spatial and keyword conditions is not achieved at the same time. Therefore, the pruning power cannot be fully utilized.

There are recent studies on integrating a spatial index with a keyword index [11, 9, 8, 20, 21, 19, 14]. The proposed methods add a keyword filter to each node in the spatial tree node that describes the keyword information in the subtree of that node. (More details are explained in Section 2.) This structure allows keyword-based pruning at the node. A filter can be implemented as an inverted list, a signature file [9], or a bitmap of the keywords in the subtree. The work in [11] constructs a global inverted index to map from keywords to tree nodes that have these keywords. These studies consider the problem of range search, nearest neighbor search [12, 17], or top-k search [8]. The work in [8] proposed two ideas to improve the performance of searching with this type of indices: using

object similarities to influence the structure of the tree index, and creating clusters for similar objects and indexing on them. The work in [20, 21] studied how to find records that are close to each other and match query keywords. The work in [19, 1] studied approximate keyword search on spatial data. The results in [11, 9, 8] show that these "integrated" index structures combining both conditions outperform the methods using two separate index structures. Therefore, in this paper, we focus on the "integrated" index structure as the baseline approach to show the improvement of our techniques.

Instant keyword search (a.k.a. interactive search, auto-complete search, or type-ahead search) has become in many search systems. Bast et al. [3, 2] proposed indexing techniques for efficient auto-complete search. The studies in [13, 6] investigated how to do error-tolerant interactive search. Li et al. [15] studied type-ahead search on relational databases with multiple tables. These studies did not focus on how to answer location-based queries. Some online map services, such as Yelp and Google Maps, recently provide location-based instant search interfaces. To our best knowledge, there are no published results about their proprietary solutions.

2 Preliminaries

In this section we formulate the problem of location-based instant search, and present a state-of-the-art approach for answering location-based keyword queries.

2.1 Data

Consider a data set of spatial keyword records. Each record has multiple attributes, including a spatial attribute A_S and a keyword attribute A_W. For simplicity, we assume that the data set has one keyword attribute, and our techniques can be extended to data sets with multiple keyword attributes. The value of the spatial attribute A_S of a record represents the geographical location of the record. The value can be a rectangle, or a point with a latitude and a longitude. The keyword attribute A_W is a textual string that can be tokenized into keywords. Figure 1 shows an example data set, and we will use it to explain the related techniques throughout this paper.

2.2 Location-Based Instant Search

A location-based instant search combines spatial search with keyword search using the AND semantics. That is, we want to retrieve records that satisfy both spatial and keyword conditions. We consider the following types of queries.

Range query: An instant range query consists of a pair $\langle R, W\rangle$, where R is a geographical area (usually represented as a rectangle or a circle), and W is a set of keywords $W = \langle w_1, w_2, \ldots, w_l\rangle$. The answer to the query is the set of records whose spatial attributes A_S geographically overlap with R, and whose keyword attributes A_W contain $w_1, w_2, \ldots, w_{l-1}$ and a keyword with w_l as a prefix[1]. For example, if we define a rectangle $R = \langle 40.776, 40.783, -73.976, -73.956\rangle$ using latitudes and longitudes (shown in the map of Figure 1 as the dashed rectangle),

[1] Our techniques can also be extended to treat all query keywords as prefixes.

the query $\langle R, \{\texttt{Christ}, \texttt{Chu}\}\rangle$ is to ask for entities located within R that have the keyword `Christ` and the prefix `Chu` in their name (e.g., record 7).

kNN query: An instant k-nearest-neighbor (kNN) query for a positive integer k is a pair $\langle P, W\rangle$, where P is a geographical point (e.g., the current location of the user), and W is a set of keywords. The answer to the query is the set of top-k records that are geographically closest to P, having the keywords $w_1, w_2, \ldots, w_{l-1}$ and a keyword with w_l as a prefix in their A_W value. For instance, the 2-NN query $\langle P, \{\texttt{Muse}\}\rangle$ asks for top-2 entities that are nearest to $P = \langle 40.786, -73.957\rangle$ (shown in the map of Figure 1), having the keyword `Muse` as a prefix in their name value (e.g., records 3 and 9).

2.3 Baseline Approach For Location-Based Keyword Search

We describe a baseline approach for answering location-based keyword queries presented in previous studies [11, 9, 8, 20]. It uses an index that extends a spatial tree index such as an R*-tree, by adding keyword filters to the tree nodes. A keyword filter at a node is a summary of the keywords in the records in the subtree of the node. We say an R*-tree node n and a keyword w are *consistent* if there exists at least one record in the subtree of node n that has keyword w. The purpose of using filters is to prune *inconsistent* branches at this node when traversing the tree to answer queries with keyword conditions. We can implement filters using multi-maps[2] from keywords to their consistent children (for this purpose, the "children" of a leaf node are its records). In the literature, the multi-values associated with a keyword w are also referred as the inverted list of w or the posting list of w.

Figure 3 presents the baseline index built on our data set. The minimum bounding rectangles (MBRs) of the tree nodes and the locations of the records are shown on the top-left map. Each node stores a multi-map from the keywords to their sets of consistent children. Notice that the multi-map is not necessarily stored in the node. For instance, in our in-memory implementation, the multi-map is stored in a separate buffer, linked to the node through a pointer. When we visit the node p when answering the query $\langle R, \{\texttt{Church}\}\rangle$, we only need to traverse the consistent children nodes (leaf c), as indicated by the keyword filter on node p. Record 7 is then retrieved from the leaf c using the keyword filter on the node, as an answer to this query. The main advantage of using this index is that we can do pruning on both the spatial condition and the keyword condition simultaneously during a search.

3 Filtering-Effective Indexing

In this section we present an index that improves the baseline index for both complete queries (in which each keyword is treated as a complete keyword) and instant queries, by using more effective filtering[3]. We first show the index

[2] A multi-map is a generalized associative array that maps keys to values. In a multi-map, multiple values can be associated for a key.

[3] This technique is orthogonal to and can be applied to the work in [8]. For simplicity we present it by improving the baseline index.

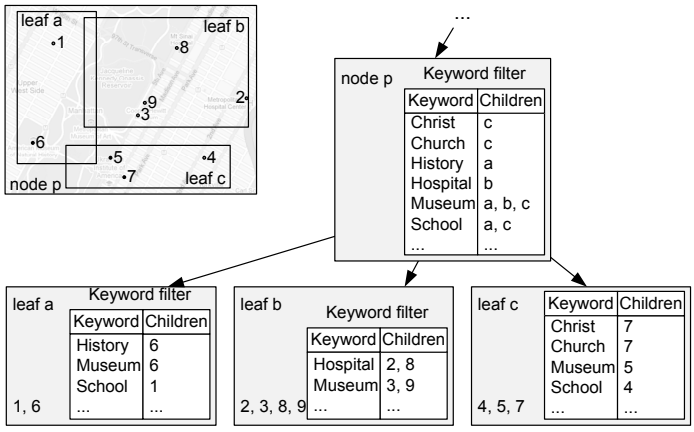


Fig. 3. A tree-based index to support location-based keyword search. The map shows the locations of the records and minimum bounding rectangles (MBRs) of the nodes.

and search technique for complete queries in this section, and then adapt it for instant queries in Section 4. We use an R*-tree as an example to explain the algorithms, since it is one of the state-of-the-art indices for geographical data. Our techniques can also be used in other tree-based indices.

3.1 Keyword Selectiveness

Our index is called filtering-effective hybrid (FEH) index. The main idea is to treat the keywords at a tree node differently when creating the filters, so that we can do direct access to records for those very selective keywords during a traversal of the R*-tree. Specifically, we define the following selectiveness es of a keyword w on a node n:

Record selectiveness: The keyword w is *record selective* on the node n, when there are only a few records in the subtree of n that have w. We use a threshold t_r to determine whether w is considered to be record selective on n, i.e., there are at most t_r records in the subtree of n that have w. In the example of Figure 3, let t_r be 1. Since there is only one record (record 7) that has the keyword `Church` in the subtree of node p, `Church` is record selective on p.

Child selectiveness: The keyword w is *child selective* on the node n, if n has at least one child that is inconsistent with w. When n is traversed to answer a query with w, only those consistent children need to be visited, and the rest can be pruned. For example, the keyword `Hospital` is child selective on the node p in Figure 3, because only the child node b of p is consistent with the keyword, and the other two children a and c can be pruned.

Non-selectiveness: The keyword w is *not selective* on the node n, if w is neither record nor child selective. In this case, the baseline index would gain no additional pruning power on n by adding w to the filter. For instance, in Figure 3, the keyword `Museum` is not selective on p, since all the children of p are consistent with the keyword (not child selective), and there are many records in the subtree of p that have the keyword (not record selective).

3.2 FEH Index

Based on the analysis above, we construct an FEH index by introducing two types of keyword filters on each tree node for record-selective keywords and child-

selective keywords. An *object filter* ("O-filter" for short) on an internal node n in the R*-tree is a multi-map from some record-selective keywords directly to the records that have these keywords. We say that such keywords are *bypassing* at node n in the subtree of n. Using an O-filter we can directly retrieve records for these record-selective keywords without traversing the subtree. The multi-values of a keyword in an O-filter can be stored as an array of the record ids with this keyword. Note that we build O-filters only on those internal nodes.

A *child filter* ("C-filter" for short) on a node n is a multi-map from some child-selective keywords to their consistent children. The C-filter is used to prune inconsistent children of n for child-selective keywords. Considering that the total number of children of n is bounded by the maximum branching factor of the node, which is often relatively small, the multi-values of the multi-map can be stored as a bit vector, in which each bit in the vector of a keyword indicates whether the corresponding child is consistent with the keyword. Notice that a non-selective keyword is absent from both filters, which implies that all children of n need to be visited. Figure 4 shows an example FEH index. On the internal node p, for instance, the C-filter contains the keywords `Hospital` and `School`, which are mapped to their consistent children nodes. The O-filter has the keywords `Christ`, `Church`, and `History`, with lists of records that have these keywords.

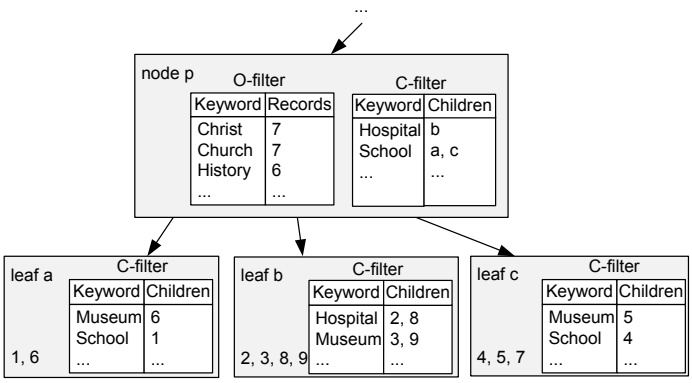


Fig. 4. An FEH index based on the tree structure of Figure 3.

In the FEH index, a keyword w is added to the O-filter and the C-filter on a node n, based on its selectiveness on n. (1) If the keyword w is record selective and not bypassing in n's ancestors, then it is added to the O-filter on n. (2) Otherwise, if w is child selective, then it is added to the C-filter on n. (3) Otherwise, w is absent in both filters on n. For instance, `Museum` is absent in the filters of node p in Figure 4, since it is not selective.

3.3 Searching with FEH Index

We present a search algorithm using an FEH index on a range query $\langle R, W \rangle$. The search algorithm for a kNN query $\langle P, W \rangle$ is similar by replacing the depth-first traversal with a priority-queue driven traversal. Using minimum distance to P as the priority.

Using a global vocabulary to check query keywords: We use a vocabulary to keep all the keywords in the data set. For each query, we first check if

each of its keywords appears in the vocabulary. The search continues only if all the query keywords are found in the vocabulary. We can also use the vocabulary to map query keywords from strings to integer ids.

Range search: We show how to do a range search with the FEH index. We recursively access tree nodes and return records that satisfy the query conditions R and W using the O-filter and the C-filter on the input node. (1) We first try to find a query keyword of W in the O-filter. If such a query keyword can be found in the O-filter, we retrieve all the records on the inverted list of this keyword, and verify whether they satisfy the keyword and spatial conditions. For those that pass the verification, we add them to the result. If we can successfully use the O-filter, we prune the whole subtree of the current node without any further processing. (2) Otherwise, if the keyword appears in the C-filter, for the current node's children that are spatially consistent with R, we prune them using the keyword's consistent children. The resulting set of children are those consistent with both the spatial condition and the keyword condition. (3) If none of the keywords in W is found in the O-filter and the C-filter, we consider all children of the current node that satisfy the spatial condition R. If the current node is an internal node, we then recursively visit its survived children nodes and retrieve answers from them; otherwise, we add all the survived records to the result.

Advantages: Compared to the baseline index, FEH can not only reduce the query time, but also reduce the index size. Specifically, (1) it can effectively prune an entire subtree if a query keyword is record selective on the root of the subtree. In Figure 4, the number of churches in the subtree of p is relatively small. When processing the query $\langle R, \{\texttt{Church}\}\rangle$ on node p, with the O-filter on p we can directly answer the query from the list of record Ids of the keyword `Church` (i.e., record 7) without visiting the subtree of p. (2) FEH reduces storage space at a node n by ignoring the keywords that are bypassing in an ancestor of n or are not selective. For example, on node p in Figure 4, `Museum` is not selective as it is consistent with all the children of p, and appears in many records in the subtree of p. We save space by not storing it in the filters on p.

4 Answering Location-Based Instant Queries

In this section we study answering location-based instant queries using the baseline index and the FEH index. We present a technique that represents the prefix filters efficiently in Section 4.1. Then we study how to compress the prefix filters in Section 4.2.

We extend the baseline index and the FEH index to answer instant queries. We first use the baseline index as an example to illustrate the techniques. The goal is to support pruning of branches that will not give answers satisfying the keyword conditions. During the search, we need to utilize a filter so that only children nodes that have a record with the query prefix are visited. We extend the baseline index by building the filters on the *consistent prefixes* as prefix filters. We say a keyword w' extends a string w, if w is a prefix of w'. A string w is a *consistent prefix* of an R*-tree node n if there exists a consistent keyword w' in the subtree of n that extends w. The prefix filter on node n maps from n's consistent prefixes to their lists of n's consistent children. For instance, on the

node p in our running example, its children nodes a and c are consistent with the prefix `Sch`. To answer a query with `Sch`, only nodes a and c are traversed.

By building filters on prefixes we can use the baseline index and the FEH index to efficiently answer instant queries, since pruning on prefix and spatial conditions happens at the same time during the search. A main challenge here is that, as prefixes (in addition to complete keywords) are added to the filters, the size of the index could dramatically increase. Like most instant search systems, it is crucial to make the index reside in memory in order to achieve a high query performance. Therefore, reducing the index size is important. Next we present techniques that efficiently store the prefix filters.

4.1 Representing Prefix Filters Efficiently

In this section we present an efficient representation of the prefix filters. A straightforward approach is to add all consistent prefixes into a prefix filter. This approach requires a lot of space. For instance, the consistent prefixes of the keyword `Hospital` on node p in Figure 3 are `H`, `Ho`, `Hos`, `Hosp`, `Hospi`, `Hospit`, `Hospita`, and `Hospital`.

Next, we show that many prefixes can be combined with each other using a radix tree [16]. The radix tree is built on *all* the keywords in the data set, with each edge of the tree labeled with one or more characters. Each node of the radix tree represents one or more prefixes in the data set. For example, Figure 5 shows the radix tree on the keywords `Christ$`, `Church$`, `History$`, `Hospital$`, `Museum$`, `Scheme$`, and `School$`. The character `$` is appended to the end of each keyword to distinguish it from the same string as a prefix, assuming it is not in the alphabet. The leftmost leaf node represents prefixes `Chr`, `Chri`, `Chris`, `Christ`, and `Christ$`. We can see that all the prefixes represented by a radix tree node have the same set of extended keywords in the dataset. Therefore, they have the same set of consistent children of an R*-tree node and records. For instance, the prefixes `Mus` and `Museu`, represented by the same radix tree node, have the same set of consistent children of p (nodes a, b, and c). We first illustrate our technique using the baseline index. Instead of adding all consistent prefixes to the filter, our technique adds consistent radix tree nodes to the prefix filter, where each radix tree node can represent multiple prefixes.

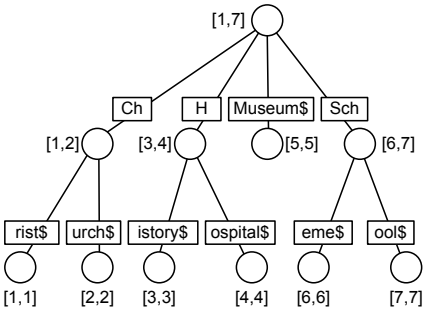


Fig. 5. Radix tree built on the global vocabulary.

To efficiently store the prefix filter on each R*-tree node, we assign numerical ids to all the keywords sorted in their lexicographical order. A prefix in the

data set can then be represented as an interval $[Id_{min}, Id_{max}]$, where Id_{min} is the minimum id of the keywords that extend the prefix, and Id_{max} is their maximum id. This representation has the following benefits. (1) All the prefixes represented by a radix tree node have the same interval. For instance, prefixes `C` and `Ch`, represented by the same node in Figure 5, have the same interval $[1, 2]$. Therefore, we can use an interval to represent a radix tree node to be stored in the prefix filter, instead of storing all the prefixes that share the same interval. (2) It is easy to test whether the node represented by one interval I_1 is an ancestor of the node represented by another interval I_2, by simply checking if I_1 contains I_2. For instance, the radix tree node for $[3, 4]$ (`H`) is an ancestor of that for $[3, 3]$ (`History$`), since the former interval contains the latter. We also say "$[3, 4]$ is a prefix of $[3, 3]$", or "$[3, 3]$ extends $[3, 4]$" interchangeably. This property is used later in the compressed prefix filter, where we need to test whether a prefix in the filter is a prefix of a query string. (3) The lexicographical order of the prefixes is defined as follows. For two intervals $I_1 = [Id_{min1}, Id_{max1}]$ and $I_2 = [Id_{min2}, Id_{max2}]$, we define:

$$\begin{cases} I_1 < I_2, \text{ if } Id_{min1} < Id_{min2} \vee (Id_{min1} = Id_{min2} \\ \qquad \wedge Id_{max1} > Id_{max2}); \\ I_1 = I_2, \text{ if } Id_{min1} = Id_{min2} \wedge Id_{max1} = Id_{max2}; \\ I_1 > I_2, \text{ otherwise.} \end{cases}$$

For example, $[3, 4] < [3, 3]$, because they tie on Id_{min}, and $[3, 4]$ has a bigger Id_{max}. This order allows us to store the intervals in a sorted array. It also allows us to use a binary search to efficiently locate the longest prefix of the query string in the array.

Within an R*-tree node, we store all its consistent prefixes in a sorted array using their interval representations and their corresponding consistent children. Figure 6 shows the prefix filter stored as sorted intervals on the R*-tree node p. The interval $[3, 4]$ for the prefix `H` is stored in the prefix filter, with its set of consistent children (nodes b and c). To answer a prefix query using the prefix filters, we can first lookup from the global vocabulary (the radix tree) to convert the query prefix to its interval representation. When accessing an R*-tree node n, we locate the corresponding interval in the prefix filter on n by performing a binary search on the sorted array of intervals, and then retrieving the consistent children nodes to visit.

FEH prefix filters: The techniques presented in Section 4.1 can be applied to the FEH index as well. Figure 7 shows the prefix filters with intervals on node p in the FEH prefix index in our running example. For instance, the record-selective prefix `Ch` is added to the O-filter on node p. It is stored as its corresponding interval $[1, 2]$, with the list of the records that have the prefix (record 7). The intervals added to a filter on the node p form a radix forest (a set of disjoint radix trees), shown on the left side of the filter in the figure.

When traversing an R*-tree node to answer a query, we first lookup the query interval from the O-filter. If the query interval is found in the O-filter, we verify the records from its list without accessing this subtree. Otherwise, we lookup the

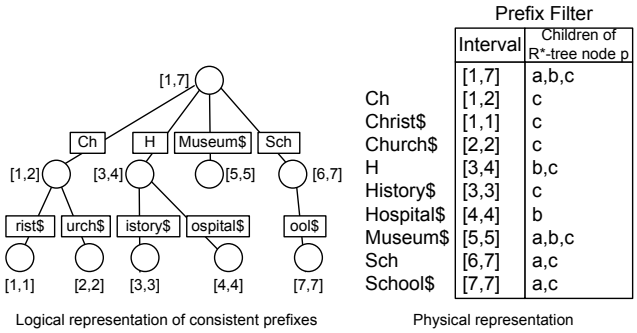


Fig. 6. The prefix filter on the R*-tree node p.

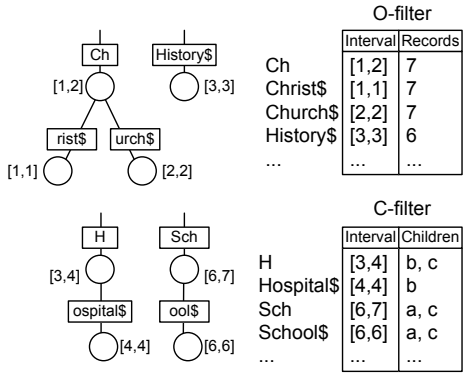


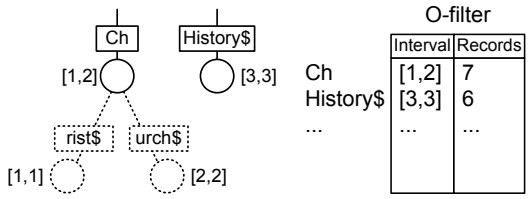
Fig. 7. Filters on node p in the FEH prefix index.

query interval from the C-filter, and visit the consistent R*-tree children. For instance, to answer the query with the prefix `Ch`, we lookup its corresponding interval $[1, 2]$ from the global vocabulary. When visiting the R*-tree node p, we lookup $[1, 2]$ from the O-filter, and then directly retrieve the records without visiting any other children nodes of p.

4.2 Compressing Prefix Filters

In this section we present a technique for further compressing the prefix filters. An interesting property of the prefix filters is the following: For two prefixes w_1 and w_2 in the same filter, if w_1 is a prefix of w_2, then the consistent set of children or records of w_1 is a superset of that of w_2. This property allows us to compress the prefix filter by removing w_2 from the filter, since we can use the superset of w_1 to answer a query of w_2, with possible false positives. Formally, we require an interval to hold the following condition in order to remain in a compressed filter on node n: the interval does not extend another interval in the same prefix filter. Otherwise, the interval can be removed from the prefix filter. For example, Figure 8(a) shows the radix forest of record-selective prefixes. $[1, 1]$ and $[2, 2]$ are removed from the compressed O-filter, as they extend another interval $[1, 2]$ in the same filter. The O-filter is physically stored as in Figure 8(b).

The technique of compressing prefix filters discussed in Section 4.2 can be utilized on both C-filters and O-filters. Here we focus on compressing the O-filters, as experiments show that the size of O-filters is much larger compared to



(a) Logical representation. Dashed nodes are removed. (b) Physical representation.

Fig. 8. Compressed O-filter on node p.

that of C-filters. Another reason is that the performance penalty is large when the compression method is applied on the C-filters, since a traversal of a query would go to branches not consistent with the query.

We can apply this technique on O-filters by enforcing the condition on some instead of all intervals to support flexible size reduction. Our experimental results show that the performance penalty on O-filters is very small even if the condition is enforced on all prefixes.

After we remove some prefixes from the filters, we need to modify the search algorithm for answering the query $\langle R, W \rangle$. The existence test of the query keyword w in the O-filter is replaced with finding a prefix w' of w in the O-filter. It can be achieved by doing a binary search on the interval array for w. Either w is returned, or the largest string w' that is smaller than w is returned. The returned w' is a prefix of w if w has a prefix in the filter. For instance, a binary search for $[2, 2]$ (`Church$`) on the interval array of the O-filter on node p returns $[1, 2]$ (`Ch`). We verify that $[1, 2]$ is a prefix of $[2, 2]$, since $[1, 2]$ contains $[2, 2]$. We use the records for $[1, 2]$ (record 7) to answer the query. The verification is performed on the records that have w' as a prefix, by checking whether they have w as a prefix. Since the set of records with w' as a prefix is a superset of the records with w as a prefix, retrieving and verifying records from the list of w' guarantees to retrieve all the answers. False positives can be eliminated by the verification. As the size of the list for w' is bounded by the record-selectiveness threshold t_r, we can answer queries using compressed O-filters with limited performance penalty, especially when t_r is selected properly. This fact is shown in our experimental evaluations.

5 Experimental Evaluation

In this section, we report our experimental results of the proposed techniques. All the algorithms (including the R*-tree) were implemented using GNU C++ and run on a Dell PC with 3GB main memory, and a 2.4GHz Dual Core CPU running a Ubuntu operating system. Our indices were created in memory using R*-trees with a branching factor of 32. The multi-map filters of all indices were implemented either using keyword ids, or prefix intervals (for instant queries). We evaluated the index size and construction time, as well as the query performances of the baseline index, and the FEH indices with various record-selectiveness thresholds t_r (denoted by FEH-t_r), for answering complete and instant queries.

5.1 Data Sets

We used two data sets in our experiments:

Business listings (Businesses). It consists of 20.4 million records of businesses in the USA from a popular directory website. We built the indices on the name, the latitude, and the longitude attributes of each business. The size of the raw data was around 4GB. The entire data set was used in the scalability tests. Since the baseline index can only handle around 5 million records in memory in some of the experiment settings, we used 5 million records of the data set for most of the experiments.

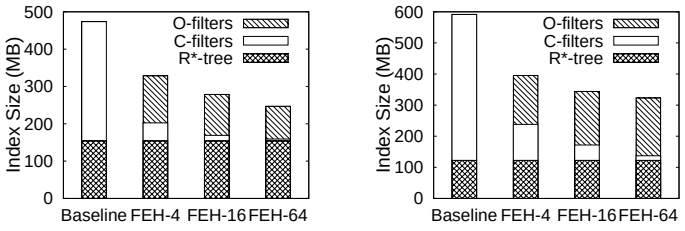
Image posts (CoPhIR) [5]. This image data set was extracted from the user posts on flickr.com. We selected 3.7 million posts that have a geographical location in the USA, and used their textual and spatial attributes. The indices were built on the title, description, tags, latitude, and longitude attributes. The size of the data was about 500MB.

We observed similar performance in our experiments on the two data sets. Due to space limitation, we mainly present the results on the Businesses data set unless otherwise noted.

5.2 Complete Keyword Search

Index Construction We constructed the baseline index and the FEH indices with different record-selectiveness thresholds t_r. The baseline index on the Business data set took 3 minutes to build, and FEH-16 took 7 minutes. The construction of FEH-16 on the Business data set used a maximum of 400MB memory.

Figure 9 shows the sizes of indices decomposed to different components on the two data sets. Indices of the same data set had the same R*-tree size as they extend the same R*-tree (e.g., 154MB for the Businesses data set). The total sizes of the filters in the FEH indices were smaller than that of the baseline index for each data set. For instance, the total size of the baseline index for Businesses was 474MB, while the total size of FEH-16 was 279MB. We also noticed that the sizes of the C-filters on the FEH indices were much smaller compared to that of the O-filters, which included the keywords as well as their corresponding lists of record ids. For both data sets, the C-filter size of FEH decreased from 48MB to 5MB as we increased the record-selectiveness threshold t_r from 4 to 64. The reason is as we increased the threshold, more keywords were qualified as record selective and bypassing and they were not added to the C-filters.

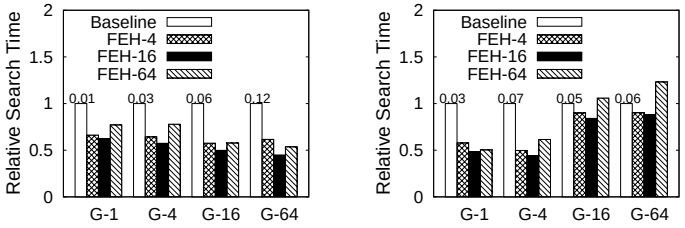


(a) Businesses (b) CoPhIR

Fig. 9. Index size for complete keyword search.

Search Performance We created query workloads by combining spatial locations with query keywords to evaluate the search performance of different indices. We manually selected 10 populous geographical areas. Each area spans 0.2 latitudes and longitudes, resulting in a 20km by 20km rectangle approximately (the exact size depends on the latitude). These areas were used as the rectangles in the range queries. The centers of these areas were used as the query points in the kNN queries, which ask for the top 10 results. To generate the queries, for each area we selected a set of keywords from its *local vocabulary*, a vocabulary for the businesses within the area, to avoid the situation where a range query returns no answer. To study the performance of queries with different keyword frequencies, we partitioned each local vocabulary into groups. For a frequency f, we used "G-f" to represent the group of keywords that appear in at least f records and in at most $4f - 1$ records. We randomly selected 100 keywords from each group, and generated 1000 location-based keyword queries for this group.

Figure 10 presents the query performance using different indices for answering range queries and kNN queries on the Businesses data set. Within each group, the query time is normalized against the time of using the baseline index. From the figures we can see that FEH indices can achieve a good speedup (up to 60%) over the baseline index for both range queries and kNN queries. The speedup was especially significant for range queries of all the groups, and for kNN queries with less frequent keywords. We can also see that the search became faster as we increased the record-selectiveness threshold from 4 to 16, and started to slow down from 16 to 64. This trend can be explained by the trade off between reduction in the number of visited nodes and increase in the number of candidate records to be verified. We observed similar trends for multi-keyword queries, due to space limitation we do not show the details.



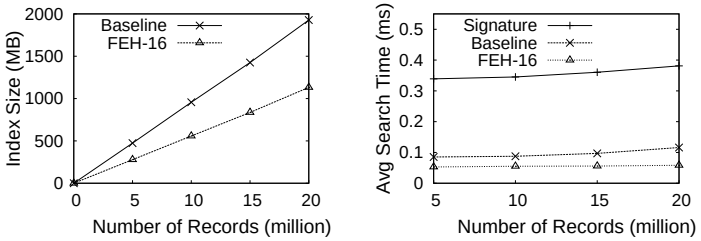
(a) Range queries (b) kNN queries ($k = 10$)

Fig. 10. 1-keyword query performance for complete keyword search. The absolute search time using the baseline index is shown on the corresponding bar in ms.

Scalability We used the Businesses data to study scalability. Figure 11(a) shows the index sizes of the baseline and FEH-16 indices when we increased the number of records from 5 million to 20 million. The size of both indices increased linearly, and FEH-16 had a smaller index size compared to the baseline index.

We conducted a scalability test of the query performance by selecting keywords from the local vocabulary for each geographical area. We randomly selected 100 keywords based on their frequencies for each geographical area, and

generated 1000 queries for each setting. We also compared the performance of our techniques to the technique in [9]. Figure 11(b) shows the average search time on range queries, using the signature-file index, the baseline index, and the FEH-16 index, as we increased the number of records. We created the signature-file indices to have the same size as their corresponding baseline indices. We can see that the search time increased sub-linearly for all the queries. The results showed that the baseline index and FEH-16 outperformed the signature-file index. This was due to the fact that the signature-file index could identify a lot of consistent children nodes as false positives, resulting in unnecessary traversal. Further, the results showed that FEH-16 is faster than the baseline index. The average search time for the baseline index increased from 0.085 ms to 0.116 ms, while that for FEH-16 increased from 0.053 ms to 0.058 ms.



(a) Index size (b) Query performance

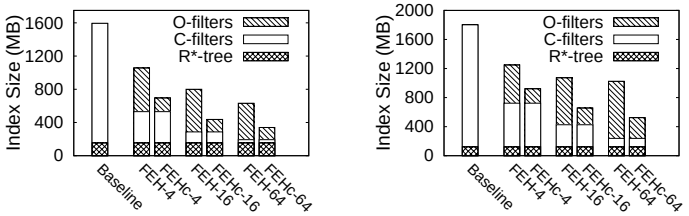
Fig. 11. Scalability for complete keyword search.

5.3 Instant Keyword Search

Index Construction We evaluated the construction and update of the baseline index and FEH for instant search. The baseline prefix index on the Business data set took 3 minutes to build, and FEH-16 on prefixes took 6 minutes. The construction of FEH-16 on prefixes used a maximum of 400MB memory on the Business data set.

Figure 12 presents the sizes of the prefix indices on the two data sets. We use "FEHc-t_r" to denote the FEH index with compressed O-filters using record-selectiveness threshold t_r. Similar trends are observed as in Figure 9. In addition, the filters on the prefix baseline indices are significantly larger compared to the filters on the non-prefix baseline indices, due to the cost of adding prefixes to the filters. We noticed that FEH and FEHc indices can significantly reduce index size. For instance, on the Businesses data set, FEH-16 reduced the index size by half (from 1.6GB to 800MB), while FEHc-16 further reduced the size by roughly half from FEH-16 (to 440MB).

Search Performance 1-keyword queries: We evaluated the search performance of the prefix indices for instant queries. We used the same methods to generate the query keywords as in Section 5.2. We generated 1000 keywords for each setting. To test prefix query performance, all prefixes of a query keyword were individually combined with the location to form the prefix queries, simulating the case where a user types character by character. We measured the

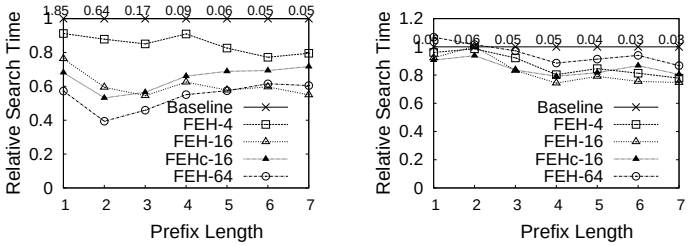


(a) Businesses (b) CoPhIR

Fig. 12. Index size for instant search.

performance of the queries with different prefix lengths. We only reported results for prefixes that appeared in at least 500 queries for each setting.

Figure 13 shows the query performance of different prefix indices for answering range queries and kNN queries. The query time was normalized against the time of using the baseline prefix index for each prefix length. The range queries with short prefix lengths were more expensive for all the indices due to the large number of results that need to be retrieved. We retrieved the top-10 results for the kNN queries. We noticed that the search-time variance for different prefix lengths was small on kNN queries compared to range queries. FEH prefix indices can improve performance of both range and kNN queries, for all the prefix lengths compared to the baseline prefix index, for up to 60%. Due to space limitation we only plotted the results FEHc-16 for FEH prefix index with compressed O-filters. The query performance penalty for compressing the O-filter was small (around 10% less improvement over the baseline index). The performance improvement of FEH indices over the baseline index on kNN queries is relatively small, due to the fact that kNN queries only retrieve the top-k answers, where k is usually very small, leaving little room for FEH to improve.

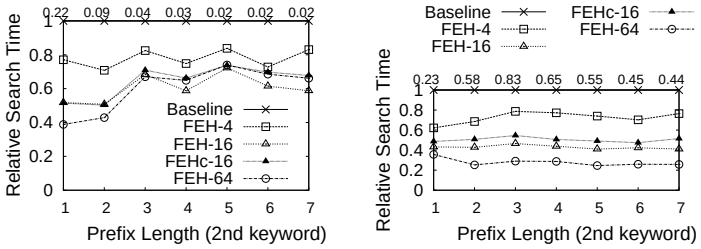


(a) Range queries (b) kNN queries

Fig. 13. 1-keyword query performance for instant search. The absolute search time using the baseline index is shown along the corresponding line in ms.

Multi-keyword queries: We also evaluated the performance of multi-keyword instant queries. To generate multi-keyword instant queries, we first randomly selected 100 sets of keywords based on their frequencies from the local co-occurrence vocabulary for each geographical area. To test the multi-keyword instant query performance, all prefixes of the last query keyword in a set were individually combined with the rest keywords as well as the location to generate the prefix queries. Figure 14 presents the performance of the 2-keyword range and kNN instant queries for different prefix lengths. Similar improvements

were observed. We observed similar trends for 3+-keyword queries, due to space limitation we do not show the details.

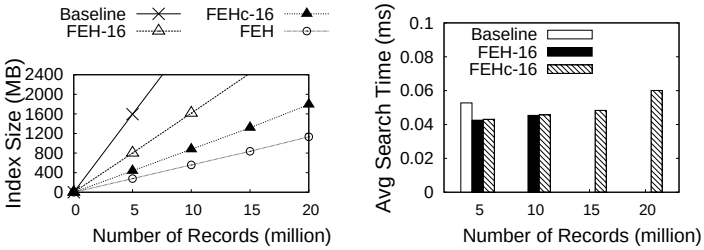


(a) Range queries (b) kNN queries

Fig. 14. 2-keyword query performance for instant search. The absolute search time using the baseline index is shown along the corresponding line in ms.

Scalability Figure 15 shows the scalability on the Businesses data set. The baseline technique on prefixes was not able to index more than 10 million records due its large index size. Similarly, FEH-16 could not index more than 15 million records. We scaled FEHc-16 all the way up to 20 million records, with the index size around 1.8GB. We also plotted the size of the FEH-16 index built on complete keywords (denoted as "FEH" in the figure) for a comparison with the prefix index sizes. It is clear that the index size increased a lot by building FEH on prefixes instead of on complete keywords, as many prefixes need to be added to the filters. Our compression technique greatly alleviated this problem.

We also studied the average kNN query performance for all prefix lengths from 1 to 7. Results in Figure 15(b) showed that the performance of FEHc-16 scaled well when we increased the data size from 5 million to 20 million.



(a) Index size (b) Query performance

Fig. 15. Scalability for instant search. "FEH" is the index for complete keywords with $t_r = 16$.

6 Conclusion

In this paper we studied location-based instant search. We first proposed an filtering-effective hybrid index (FEH) that judiciously uses two types of keyword filters based on their selectiveness to do powerful pruning. We then developed indexing and search techniques that utilize the FEH index and store prefix information to efficiently answer instant queries. Our experiments demonstrated the high efficiency and scalability of our techniques.

Acknowledgements: We thank Sattam Alsubaiee for his discussions in this work.

References

1. S. M. Alsubaiee, A. Behm, and C. Li. Supporting location-based approximate-keyword queries. In *ACM SIGSPATIAL GIS*, 2010.
2. H. Bast, C. W. Mortensen, and I. Weber. Output-sensitive autocompletion search. In *SPIRE*, pages 150–162, 2006.
3. H. Bast and I. Weber. Type less, find more: fast autocompletion search with a succinct index. In *SIGIR*, pages 364–371, 2006.
4. 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.
5. 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.
6. S. Chaudhuri and R. Kaushik. Extending autocompletion to tolerate errors. In *SIGMOD Conference*, pages 707–718, 2009.
7. Y.-Y. Chen, T. Suel, and A. Markowetz. Efficient query processing in geographic web search engines. In *SIGMOD*, 2006.
8. G. Cong, C. S. Jensen, and D. Wu. Efficient retrieval of the top-k most relevant spatial web objects. *PVLDB*, 2(1), 2009.
9. I. D. Felipe, V. Hristidis, and N. Rishe. Keyword search on spatial databases. In *ICDE*, 2008.
10. A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, 1984.
11. R. Hariharan, B. Hore, C. Li, and S. Mehrotra. Processing spatial-keyword (SK) queries in geographic information retrieval (GIR) systems. In *SSDBM*, 2007.
12. G. R. Hjaltason and H. Samet. Ranking in spatial databases. In *SSD*, pages 83–95, 1995.
13. S. Ji, G. Li, C. Li, and J. Feng. Efficient interactive fuzzy keyword search. In *WWW*, pages 371–380, 2009.
14. A. Khodaei, C. Shahabi, and C. Li. Hybrid indexing and seamless ranking of spatial and textual features of web documents. In *DEXA*, pages 450–466, 2010.
15. G. Li, S. Ji, C. Li, and J. Feng. Efficient type-ahead search on relational data: a TASTIER approach. In *SIGMOD*, 2009.
16. D. R. Morrison. Patricia - practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, 1968.
17. N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *SIGMOD Conference*, pages 71–79, 1995.
18. S. Vaid, C. B. Jones, H. Joho, and M. Sanderson. Spatio-textual indexing for geographical search on the web. In *SSTD*, 2005.
19. B. Yao, F. Li, M. Hadjieleftheriou, and K. Hou. Approximate string search in spatial databases. In *ICDE*, 2010.
20. 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*, 2009.
21. D. Zhang, B. C. Ooi, and A. K. H. Tung. Locating mapped resources in web 2.0. In *ICDE*, 2010.
22. Y. Zhou, X. Xie, C. Wang, Y. Gong, and W.-Y. Ma. Hybrid index structures for location-based web search. In *CIKM*, 2005.