Location-Aware Instant Search

Ruicheng Zhong[†], Ju Fan[†], Guoliang Li[†], Kian-Lee Tan[‡], Lizhu Zhou[†]
[†]Department of Computer Science and Technology, Tsinghua National Laboratory for Information Science and Technology (TNList), Tsinghua University, Beijing, China
[‡]Department of Computer Science, School of Computing, National University of Singapore, Singapore
zrc1101001@gmail.com; fan-j07@mails.thu.edu.cn;
{liguoliang,dcszlz}@tsinghua.edu.cn; tankl@comp.nus.edu.sg

ABSTRACT

Location-Based Services (LBS) have been widely accepted by mobile users recently. Existing LBS-based systems require users to type in complete keywords. However for mobile users it is rather difficult to type in complete keywords on mobile devices. To alleviate this problem, in this paper we study the location-aware instant search problem, which returns users location-aware answers as users type in queries letter by letter. The main challenge is to achieve high interactive speed. To address this challenge, in this paper we propose a novel index structure, prefix-region tree (called PR-Tree), to efficiently support location-aware instant search. PR-Tree is a tree-based index structure which seamlessly integrates the textual description and spatial information to index the spatial data. Using the PR-Tree, we develop efficient algorithms to support single prefix queries and multi-keyword queries. Experiments show that our method achieves high performance and significantly outperforms state-of-the-art methods.

Categories and Subject Descriptors

H.2.8 [**Database Applications**]: Spatial databases

General Terms

Algorithms, Design, Experimentation, Performance

Keywords

Type-ahead search, Keywords search, Spatial databases

1. INTRODUCTION

Location-Based Services (LBS) have become more and more popular and attracted significant attention from both academic and industrial communities recently. Many LBS based systems, such as AT&T Location Information Services[1], have been deployed to provide users with location-aware experiences, and they are widely accepted by millions of mobile users, thanks to the modern mobile devices.

[1]http://www.wireless.att.com/lbs/

CIKM'12, October 29–November 2, 2012, Maui, HI, USA.

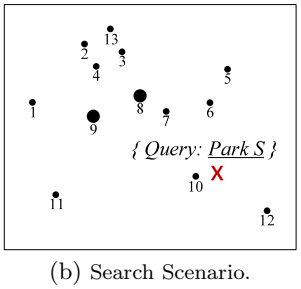
Most of the existing studies adopt a *spatial keyword search* based method to help users retrieve location-aware answers [7, 5]. Given a set of objects with spatial information and textual description (e.g., points-of-interest (POIs)) and a user query with location and keywords, spatial keyword search finds top-k relevant objects by considering the distance and textual relevance between the query and objects. For example, if a user wants to find a gas station nearby, she can issue a keyword query "`gas station`" to a LBS system, which returns the relevant gas stations by considering the user's location and keywords.

Traditional spatial keyword search method requires users to type in *complete* keywords for finding location-aware answers. However for mobile users, typing a complete keyword is tedious and also susceptible to errors. To alleviate this problem, instance search (also known as type-ahead search or search-as-you-type) [1, 17, 16, 3, 15, 14, 18] is proposed to provide users with new search experiences, which returns relevant answers as users type in queries letter by letter. Recently many systems, e.g., Google, have been deployed to support instant search.

It is very natural to extend instant search to support spatial keyword search. To this end, in this paper we study the location-aware instant search problem. In our method, as a user types in queries letter by letter, the system returns the location-aware answers on-the-fly, and provides the user with instant replies. Figure 1 provides an example of location-aware instant search over 13 POIs. At every keystroke a user types in our system, the system takes her current input string as a query and returns the relevant location-aware answers instantly. For instance, as shown in Figure 1(b), when the user types in a partial query "`park s`", the nearest objects containing complete keyword "`park`" and the prefix "`s`", i.e., o_9 and o_8, are returned. Obviously this new search method can help users to find desired answers in a more friendly way than the traditional methods.

ID	Name	Location
1	Stadium	(41.754, -76.779)
2	Palace Street	(42.434, -75.975)
3	Pavement	(42.265, -75.582)
4	Stephan Park	(42.187, -75.818)
5	Shipyard	(42.188, -73.983)
6	Stock	(41.735, -74.221)
7	Parliament	(41.623, -74.819)
8	Studio Park	(41.834, -75.126)
9	Skydive Park	(41.508, -75.809)
10	Police	(40.799, -74.378)
11	Spring	(40.684, -76.312)
12	Post	(40.457, -73.462)
13	Station	(42.761, -75.674)

(a) Data Sample.



(b) Search Scenario.

Figure 1: An example for location-aware instant search on query "`park s`" and located at "×".

It is rather challenging to support location-aware instant search due to the requirement of a high interactive speed. Existing studies [13, 21] devise hybrid index structures to address this challenge. However, [13] has very limited filtering power for short or frequent query prefixes. [21] cannot support multi-keyword queries and it also consumes a large amount of memory. Moreover, both of them cannot perform well for large data sets since they fail to fully utilize the textual and spatial pruning simultaneously.

To address these limitations, in this paper, we propose a novel index structure, prefix-region tree (called PR-Tree), to efficiently support location-aware instant search. PR-Tree is a tree-based index structure. Different from traditional indices, PR-Tree considers both textual partitioning and spatial partitioning simultaneously to build the index. Thus it can achieve great improvement of efficiency for location-aware instant search.

Our contributions are summarized as follows.

- We propose a novel data structure, called PR-Tree. It seamlessly integrates the spatial information and textual description to index the spatial data.
- Using our PR-Tree index structure, we develop effective algorithms to efficiently answer single prefix queries and multi-keyword queries.
- We have conducted thorough experiments and the experimental results show that our method achieves high performance and significantly outperforms state-of-the-art methods.

The rest of this paper is organized as follows. We formulate the problem of location-aware instant search and discuss related work in Section 2. We introduce our proposed index structure PR-Tree in Section 3. We develop efficient search algorithms in Section 4. In Section 5, we discuss how to support sophisticated ranking functions in PR-Tree. Experiments are provided in Section 6. Finally, we conclude the paper in Section 7.

2. PRELIMINARIES

In this section, we first formalize the problem of location-aware instant search in Section 2.1, and review related work in Section 2.2.

2.1 Problem Definition

Data Model. Our work considers a set of objects, $\mathcal{O} = \{o_1, o_2, \ldots, o_{|\mathcal{O}|}\}$ in a spatial database. Each object $o \in \mathcal{O}$ consists of spatial information $o.l$ and textual information $o.W$, denoted by $o = (l, W)$. Specifically, the spatial information $o.l$ represents a location in the two-dimensional geographic space, and the textual information $o.W$ is a set of distinct words, denoted by $o.W = \{w_1, w_2, \ldots w_{|o.W|}\}$. For example, Figure 1 shows 13 objects, each of which has a location and a set of words, e.g., $o_2 = (l_2, \{\texttt{palace}, \texttt{street}\})$.

Query Model. Our paper focuses on supporting location-aware instant query q that consists of a location $q.l$, a set of *complete* keywords $q.W = \{w_1, w_2, \ldots, w_{|q.W|}\}$ that a user has typed, a prefix $q.p$ that the user is typing in and an integer k, denoted by $q = (l, W, p, k)$. Given a set of objects $\mathcal{O}$, the answer of q, denoted by $\mathcal{R}$, is the k *nearest* objects sorted by their distances to query location $q.l$, where each object o satisfies

(1) object o contains all complete keywords in the query, i.e., $o.W \supseteq q.W$, and
(2) o has at least one word with $q.p$ as its prefix, i.e., $\exists w \in o.W, p \preceq w$, where $p \preceq w$ denotes that p is a prefix of w.

We assume that $\forall w_i, w_j \in q.W$ that $w_i(w_j)$ is not a prefix of $w_j(w_i)$. Let $\texttt{dist}(q, o)$ represent the distance between query location $q.l$ and object location $o.l$. In this paper, we use Euclidean distance as the distance measurement.

Problem Formulation. Based on these notations, we formulate the location-aware instant search problem.

Definition 1 (Location-Aware Instant Search). *Consider a set of objects $\mathcal{O} = \{o_1, o_2, \ldots, o_{|\mathcal{O}|}\}$ and a location-aware instant query $q = (l, W, p, k)$. It returns the top-k objects $\mathcal{R} \subseteq \mathcal{O}$ such that each object $o \in \mathcal{R}$ satisfies $o.W \supseteq q.W$ and $\exists w \in o.W, p \preceq w$, sorted by their distances to the query location.*

To make it consistent with existing work [13, 21], we only use the distance to rank the answers. We will discuss how to support other complex ranking functions in Section 5.

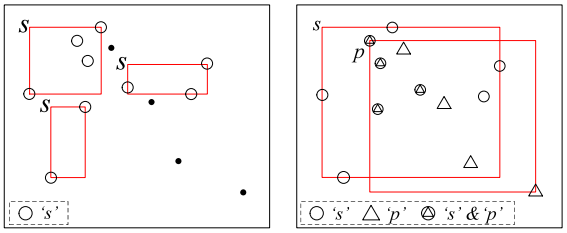
2.2 Related Work

Location-Aware Instant Search: Existing studies devised different index structures [13, 21] to support location-aware instant search.

Ji et al. [13] proposed an R-tree based method called Filtering-Effective Hybrid Indexing (FEH). The method first builds an R-tree on top of locations of all objects. Then, in each R-tree node, it adds textual filters with possible prefixes contained by the objects in this node. Given a query, it traverses the R-tree from the root to the leaves using the best-first-search strategy [10]. Whenever a tree node is reached, it first employs the filter to decide whether to insert its child nodes or objects into a priority queue. The search stops when the first k answers have been found. However, the FEH algorithm is very expensive for short and frequent prefixes, since the number of the potential candidate objects is extremely large, and the filters have poor pruning power.

Materialized Trie (MT) is a trie based method [21]. First, it builds a trie structure on top of words of all objects. Each leaf node has an inverted list to record those objects which contain the corresponding word. To incorporate spatial pruning power into the trie structure, spatial information is added in each trie node. For a trie node with a prefix p, it divides the whole query space into R regions and maintains an upper-bound score for each region to indicate that a query with prefix p will reach the maximal score in this region. However it is rather expensive to materialize the information for all trie nodes. Thus it proposes to select M nodes for materialization. The main problem for MT is that it will consume a large amount of memory. To achieve high performance, it needs to divide the space into a large number of smaller regions, thus incurs significant of space utilization. Another limitation of MT is that it cannot support multi-keyword queries, which is common in many applications.

Spatial Keyword Search: Spatial keyword search has been widely studied in the database community [24, 4, 9, 7, 5, 23, 22, 20, 19, 6, 11]. Given a set of points-of-interest (POIs) and a query with keywords and location, the spatial keyword search aims to find the relevant POIs by considering both spatial proximity and textual relevance. [24, 9, 4]



(a) Spatial partition for the same prefix "s". (b) Textual partition for a specific region.

Figure 2: Spatio-textual partition.

solved the range query problem by the use of R-tree. Felipe et al. [7] integrated signature files into R-tree. Cong et al. [5] proposed the IR-tree by combining the inverted files and R-tree. Yao et al. [23] studied how to support approximate string matching. Wu et al. [22] studied spatial keyword search for moving objects and Lu et al. [20] tracked reverse spatial and textual k nearest neighbor search.

Traditional spatial keyword search method requires users to type in *complete* keywords to find location-aware answers. In this paper, we study a more user-friendly search method, location-aware instant search. The method returns relevant POIs on-the-fly when users type in keywords letter by letter, which helps users find information with less typing efforts.

3. PREFIX-REGION TREE

In this section, we introduce a novel index structure, called prefix-region tree, to support efficient location-aware instant search. We first introduce our basic idea in Section 3.1 and give an overview of the prefix-region tree in Section 3.2. Then, we respectively discuss the tree construction and update in Section 3.3 and Section 3.4. Finally we discuss some issues on PR-Tree in Section 3.5.

3.1 Basic Idea

We can extend existing textual index structures (e.g., trie) or spatial index structures (e.g., R-Tree) to support location-aware instant search. Using a trie index based on the textual information of objects, we can adopt a *text-only* strategy to obtain the top-k answers as follows. Given a query q, we first retrieve the objects which satisfy the textual constraints, i.e., such $o \in \mathcal{O}$ that $o.W \supseteq q.W$ and $\exists w \in o.W, q.p \preceq w$, and then sort them in ascending order by their distances to $q.l$ to get the top-k answers. Obviously, this method is inefficient since it never considers the spatial pruning, and may involve a huge number of objects which are not the top-k answers.

On the other hand, using an R-tree index based on the spatial information of objects, we can employ a *spatial-only* strategy. Specifically, we can adopt the best-first traversal method [10] to iteratively find the nearest objects. Then, we examine whether the objects satisfy the textual constraints mentioned above. However, this method fails to consider the textual pruning since the traversal over the R-tree ignores the words of the underlying objects.

To address these problems, we aim to build a more effective index structure which integrates both the textual and spatial information seamlessly. To achieve this goal, we have the following observations.

(1) Given a specific prefix p, the objects with p may scatter in different areas. Based on the spatial distribution, we can partition the objects into several regions to facilitate effective spatial pruning. Figure 2(a) shows objects containing prefix "s" (represented by circles), which can be partitioned into three regions. Using these regions, we can estimate objects' distance bounds to the query location, and utilize spatial pruning to efficiently obtain top-k answers.

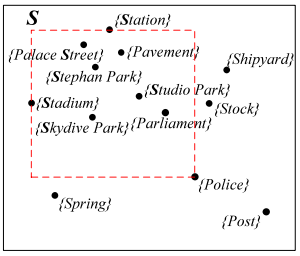


Figure 3: Prefix region.

(2) Given a specific region r, the objects in r may contain different prefixes but it is hard to distinguish them by spatial information only. Thus, we can partition the objects based on the prefixes, in order to facilitate effective textual pruning. Figure 2(b) provides some objects in a region. We can partition the objects based on different prefixes they contain, e.g., "s" and "p", and efficiently prune the objects dissatisfying the textual constraints.

If we partition the objects by simultaneously using the textual and spatial information, we can fully utilize the spatial textual pruning at query time. However it is non-trivial to devise an index structure that seamlessly combines textual and spatial partitions. To address this challenge, we propose the prefix-region tree in the following sections.

3.2 Prefix-Region Tree

For ease of presentation, we first introduce an important concept, called *prefix region*. Intuitively, a prefix region consists of a prefix for textual partition and a region for spatial partition. Formally, the prefix region is defined as follows.

DEFINITION 2 (PREFIX REGION). *Let p denote a prefix and r denote an* MBR*(Minimum Bounding Rectangle). A prefix region is defined as a combination of prefix p and region r, denoted by $f = \langle p, r \rangle$.*

Given a prefix region f, we use $\mathcal{O}_f$ to represent the objects satisfying f, that is, each object $o \in \mathcal{O}_f$ has prefix $f.p$ and its location $o.l$ is within region $f.r$. Take Figure 3 as an example. Consider a prefix region $f = \langle$"s"$, r\rangle$ and a set of objects $\mathcal{O} = \{o_1, o_2, o_3, o_4, o_7, o_8, o_9, o_{10}, o_{13}\}$. We have $\mathcal{O}_f = \{o_1, o_2, o_4, o_8, o_9, o_{13}\}$ since all objects in $\mathcal{O}_f$ have prefix "s" and their locations are within r, as illustrated by the dashed-line rectangle labelled with prefix "s" in Figure 3. In addition, we define that a prefix region f_1 is *subsumed* by another prefix-region f_2, if and only if 1) $f_2.p$ is a prefix of $f_1.p$, denoted by $f_2.p \preceq f_1.p$, and 2) $f_2.r$ encloses $f_1.r$, denoted by $f_1.r \subseteq f_2.r$.

Then, we define the prefix-region tree (or PR-Tree for short) as follows. PR-Tree organizes all objects in $\mathcal{O}$ in a hierarchical manner, where each node is a prefix region[2]. In the PR-Tree, each non-leaf node subsumes all its child nodes. Each leaf node, say $f_l = \langle p_l, r_l \rangle$ is associated with an object list, denoted by `ObjectList`, which contains the objects with prefix p_l located within region r_l. In particular, the root is $f_{root} = \langle \emptyset, r_{all} \rangle$, where r_{all} represents the `MBR` of all objects in $\mathcal{O}$. Obviously, in a PR-Tree, the region sizes of nodes in lower levels are smaller than those in higher levels.

[2]For ease of presentation, we use "prefix region" and PR-Tree node interchangeably.

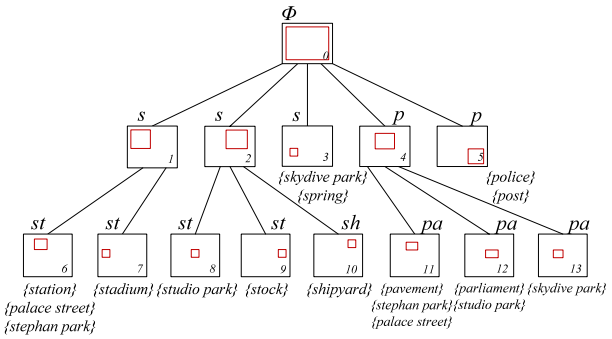


Figure 4: A PR-tree.

In order to avoid large tree depths, we introduce a parameter M to allow at most M objects contained in every leaf. Figure 4 provides an example of a PR-Tree. The string on the top-left of the node box denotes the prefix $f.p$, and the inner frame denotes the region $f.r$.

3.3 Tree Construction

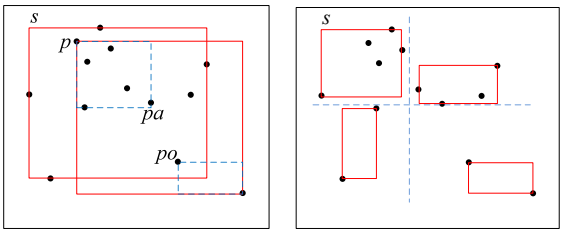
We construct a PR-Tree in a top-down manner. We first initialize the root node $f_{root} = \langle \emptyset, r_{all} \rangle$ with object set $\mathcal{O}$. Then, we split $\mathcal{O}$ into subsets to generate child nodes. Similarly, we recursively split the generated nodes until the number of objects in each leaf node is not greater than M. Obviously, the essential task is how to split a node f. To this end, we consider both textual and spatial partitioning.

Textual partitioning groups the objects $\mathcal{O}_f$ of node f based on the textual information. Specifically, consider a set of words W_f with prefix $f.p$ contained by the objects in $\mathcal{O}_f$. Based on W_f, we can construct a radix trie[3] where each path from the root to a node corresponds to a prefix. Using the radix trie, we can define `Next`$(f.p)$ as the child nodes of $f.p$ in the trie. For example, given a set of keywords W_f ={`pavement, palace, park, parliament, police, post`}, we have `Next`("`p`") ={`pa, po`} and `Next`("`pa`")= {`pavement, palace, par`}. Based on prefixes in `Next`$(f.p)$, we can partition objects $\mathcal{O}_f$ into subsets for generating child nodes in the PR-Tree. Figure 5(a) provides an example of textual partitioning. The root node "$\emptyset$" is partitioned into two child nodes respectively with prefixes "`s`" and "`p`" in `Next`$(\emptyset)$. Then, node with prefix "`p`" is further partitioned into two child nodes respectively having "`pa`" and "`po`".

Spatial partitioning groups objects $\mathcal{O}_f$ of node f based on the spatial information. Specifically, we partition region $f.r$ into subregions, each of which contains subsets of $\mathcal{O}_f$. In this paper, we consider one of the most fundamental strategies, i.e., to partition one region into four non-overlapped subregions by splitting at a central point horizontally and vertically. We omit discussing other partitioning methods in this paper due to the space limitation.

More formally, we partition region $f.r$ into four regions, r^{ll} (lower left), r^{ul} (upper left), r^{lr} (lower right) and r^{ur} (upper right), and respectively represent the object sets in the regions as $\mathcal{O}_f^{ll}$, $\mathcal{O}_f^{ul}$, $\mathcal{O}_f^{lr}$ and $\mathcal{O}_f^{ur}$. A straightforward method is to simply partition r into four regions of the same size, which is used in a quad-tree [8]. However, this method may result in skew distribution of objects, that is, some regions may contain more objects than the others. To make the partitioning as even as possible, i.e., each subset contains nearly the same number of objects, we select the *centroid* as the center for spatial splitting. As shown in Figure 5(b), the objects are partitioned into four sub-regions. Note that the partition strategies may produce less than four sub-regions, since some regions contain no object.

[3]A radix trie is a space-optimized trie structure where each node with only one child is merged with its child.



(a) Textual Partition. (b) Spatial Partition.

Figure 5: Textual partition and spatial partition.

Based on the textual and spatial partitioning, we present the algorithm for constructing a PR-Tree, as shown in Algorithm 1. The algorithm first initializes a root node $f_{root} = \langle \emptyset, r_{all} \rangle$, and puts the root with its corresponding objects $\mathcal{O}$ into a stack. Then, the algorithm iteratively employs the following textual and spatial partitioning operations on each node f, which is illustrated in Figure 6.

1) The algorithm employs textual partitioning to obtain several *intermediate* nodes, i.e. $\langle p_1, r_{p_1} \rangle, \langle p_2, r_{p_2} \rangle, \ldots, \langle p_k, r_{p_k} \rangle$, where $p_i \in$ `Next`$(f.p)$.

2) For each intermediate node $\langle p_i, r_{p_i} \rangle$, the algorithm employs spatial partitioning to further generate four nodes, $\langle p_i, r^{ll}_{O_{p_i}} \rangle$, $\langle p_i, r^{ul}_{O_{p_i}} \rangle$, $\langle p_i, r^{lr}_{O_{p_i}} \rangle$ and $\langle p_i, r^{ur}_{O_{p_i}} \rangle$. These nodes are then assigned to be the children for the node $\langle p, r_O \rangle$.

This above node-splitting procedure stops when every leaf node contains no more than M objects.

Algorithm 1: ConstructPRTree ($\mathcal{O}$, M)

```
Input: O: An object set; M: a parameter
Output: T: A constructed PR-Tree
1  begin
2    Initialize a PR-Tree T.root ← ⟨∅,r_O⟩ ;
3    Initialize a stack, Stack ← ∅ ;
4    Stack.Push(⟨ T.root, O ⟩ ) ;
5    while Stack not empty do
6      ⟨ n, O_n ⟩ ← Stack.Pop() ;
7      if |O_n| ≤ M then
8        n.ObjectList ← O_n ;
9      else
10       C_t ← TextPartition (n.p, O_n) ;
11       foreach c ∈ C_t do
12         C_ts ← SpatialPartition (c) ;
13         n' ← CreateTreeNode ( C_ts ) ;
14         Stack.Push( ⟨n', C_ts⟩ ) ;
15 end
```

3.4 Tree Update

In this section, we discuss the update operations, i.e., insertion and deletion of the PR-Tree.

3.4.1 Insertion

We discuss an algorithm to insert an object $o = (l, W)$ into a PR-Tree. The basic idea is to traverse the PR-Tree and select an appropriate node for insertion. Specifically, for every word w_i in $o.W$, the algorithm traverses the PR-Tree from the root. For a visited node f, the algorithm identifies

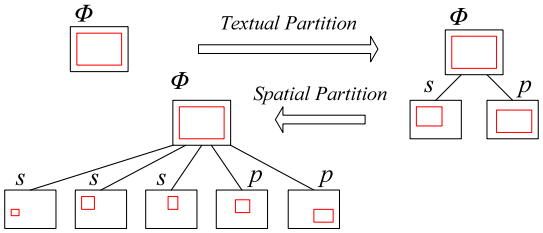


Figure 6: PR-tree construction.

the nearest child node f_c that $f_c.p \preceq w_i$ with respect to $o.l$. After selecting the child node f_c, the algorithm updates f_c as follows. If $o.l$ is not contained within region $f_c.r$, the algorithm enlarges region $f_c.r$ to a region containing $o.l$. We can recursively apply the above procedure until reaching a leaf node f_l. If f_l contains less than M objects, we simply insert o into f_l's object list. Otherwise, we employ the textual and spatial partitioning in Section 3.3 to further split f_l, and insert o into a new leaf node.

The time complexity scales to the height of the PR-Tree tree. Formally, it takes $\mathcal{O}(\log N)$ to finish an insertion operation, where N is the number of objects.

3.4.2 Deletion

We propose a bottom-up algorithm for deleting an object from a PR-Tree. When deleting an object o, the algorithm first finds all the leaf nodes which containing o, and removes it from the object list in the leaf nodes. Then, it updates the intermediate nodes of the PR-Tree in a bottom-up way. The basic idea is to examine whether the `MBR`s of intermediate nodes are affected by the object deletion. Specifically, if $o.l$ is located at the border of the `MBR` of a given intermediate node, the `MBR` needs to be re-calculated based on the `MBR`s of its child nodes. The algorithm terminates when reaching the root of the PR-Tree.

Note that, according to our tree construction, a node must have at least two child nodes. Therefore, if a node f is empty after deletion, we examine the number of its child nodes for the parent node of f. If f is an only child, we merge the node f to its parent node.

Then we analyse the time complexity. Since a deletion operation needs to examine $|o.W|$ paths from leaves to root, the time complexity is bounded to $O(|o.W| \log N)$.

3.5 Discussion on PR-Tree

In this section, we discuss three issues on PR-Tree. The first is on the balancing issue, the second is on the space complexity and the third is on the storage alternatives.

3.5.1 Balancing Issue

PR-Tree is not a balanced tree, since the textual partitioning cannot guarantee that each child node contains the same number of objects[4]. However, it does not affect the searching efficiency on the PR-Tree due to the following reasons.

1) The height of our PR-Tree is not very large since the fanout of each intermediate node is large. Let Σ denote the alphabet generating all words. The fanout of each intermediate node is at most $|\Sigma| \times 4$, which is usually large. We constructed PR-Trees on two real datasets in our experiments, and the heights were respectively 6.35 and 5.29.

2) There are no rather long branches due to our spatial partitioning strategies. Since we choose centroid as the center for spatial partitioning, the descendant nodes would contain roughly the same number of objects, leading to the maximum length of branches scaling to $\log N$. For example, in our experiments, the maximum branch lengths of the PR-Trees on the two datasets were respectively 12 and 14.

Therefore, although the PR-Tree is not balanced, the time of traversing different branches would differ insignificantly.

[4]It is similar to a radix-tree since we employ `Next()` function.

3.5.2 Space Complexity

In this section, we analyse the space complexity of the PR-Tree. Recall that the partitioning process of a node terminates when it contains at most M objects. Therefore, we can prove that the upper bound of the total number of leaf nodes in a PR-tree is within $[\frac{\sum |o_i.W|}{M}, \sum |o_i.W|]$, as shown in Lemma 1.

LEMMA 1. *The bound for the total number of leaf nodes in a PR-tree is* $[\frac{\sum |o_i.W|}{M}, \sum |o_i.W|]$

PROOF. We first prove that any object will appear in at most $|o.W|$ leaf nodes of the PR-tree. According to the building process, the regions for the nodes with the same prefix will never be overlapped. Each word of the objects in $\mathcal{O}$ will finally be included in one leaf node, and hence an object o is contained in at most $|o.W|$ leaf nodes. Thus the upper bound is $\sum |o_i.W|$.

On the other hand, each leaf node contains M objects, thus the lower bound is $\frac{\sum |o_i.W|}{M}$. □

Based on Lemma 1, the indexing size is scalable to the volume of the data set, i.e., $\sum |o.W|$.

3.5.3 Storage Alternatives

In this paper, we only consider maintaining the PR-Tree in memory. Nevertheless, PR-Tree is feasible to be disk-based. Since each non-leaf node only needs to store a few number of attributes, i.e., prefix, `MBR` and child list, and each leaf node needs to store the `ObjectList`, the information on one node is possible to be stored in one page of disk. However, in this paper, we mainly focus on devising the PR-Tree in memory and demonstrate its supreme efficiency, while taking the disk-based alternative as future work.

4. SEARCH ALGORITHMS

In this section, we first propose an algorithm for single-prefix queries in Section 4.1, and then extend the algorithm to support multi-keyword queries in Section 4.2.

4.1 Algorithm for Single-Prefix Queries

We first consider a single prefix query q with empty complete-keyword set, i.e., $q.W = \emptyset$. Given the query, we employ the effective best-first traversal algorithm (BFT) [10], to efficiently retrieve the top-k answers using the PR-Tree. The BFT algorithm uses a priority queue for maintaining the nodes and objects in the PR-Tree that need to be visited. For each node f in the queue, we compute the minimum distance between query location $q.l$ and the corresponding `MBR` $f.r$, denoted by `MIND`$(q.l, f.r)$. For each object o in the queue, we compute its distance to $q.l$, i.e., `dist`(q, o). Then, we sort the elements in the queue in ascending order by their minimum distances.

Due to the space limitation, we omit the details for BFT and only show an example of Algorithm 2 in this section.

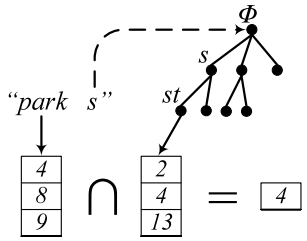


Figure 7: Multi-keyword query processing.

EXAMPLE 1. *Consider the objects in Figure 1 and the PR-Tree in Figure 4. For a query* $q = \{(40.5, -74.0), \emptyset,$ *'p',* $2\}$*, we compute top-2 answers as follows.*
Step 1*: Enqueue root node with* $\langle \emptyset, 0 \rangle$*.*
Queue: $\boxed{\langle \emptyset, 0 \rangle}$
Step 2*: Dequeue the top element* $\langle \emptyset, 0 \rangle$*, expand its child nodes containing prefix 'p', calculate their minimal distances to q.l and enqueue them.*
Queue: $\boxed{\langle p, 0 \rangle}\boxed{\langle p, 1.19 \rangle}$
Step 3*: Dequeue the* $\langle p, 0 \rangle$*, expand all objects containing prefix 'p', and enqueue them.*
Queue: $\boxed{\langle o_{10}, 0.38 \rangle}\boxed{\langle o_{12}, 0.54 \rangle}\boxed{\langle p, 1.19 \rangle}$
Step 4*: Dequeue* $\langle o_{10}, 0.38 \rangle$*. Since it is an object, we directly put it into the result list. So does* $\langle o_{12}, 0.54 \rangle$*. Now, top-2 answers have been retrieved. Algorithm terminates.*

Algorithm 2: kNNQuery (p, l, k, T)

Input: p : Prefix; l: Location; k: Number; T: PR-Tree
Output: $\mathcal{R}$: The top-k result list

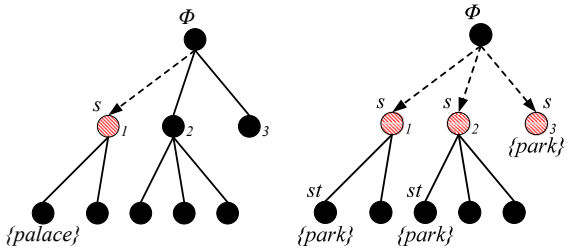
```
1  begin
2      Initialize a priority queue Q ← ∅ ;
3      Q.Enqueue(⟨ T.root, 0⟩) ;
4      while R.Size() < k & Q not empty do
5          n ← Q.Dequeue() ;
6          if n is an object then Insert n into R ;
7          else if n is a leaf node then
8              foreach Object o contains prefix p do
9                  Q.Enqueue(⟨ o, dist(l, o.l)⟩);
10         else if n is an intermediate node then
11             foreach Child node c ∈ of n do
12                 if c.p ⪯ p or p ⪯ c.p then
13                     Q.Enqueue(⟨ c, MIND(l, c.r)⟩) ;
14 end
```

4.2 Algorithm for Multi-Keyword Queries

In this section, we extend our algorithm to support queries with multiple keywords, i.e., $q.W \neq \emptyset$. We first introduce an intersection-based method and then propose a cost-based method to improve the performance.

4.2.1 An Intersection-based Method

To support multi-keyword queries by PR-Tree, a straightforward method is to simply extend the algorithm for single-prefix queries. Specifically, consider a query $q = (l, W, p, k)$ where $q.W \neq \emptyset$. Each time when a dequeued element is a leaf node, we can only enqueue the objects which contain all keywords in $q.W$. To efficiently examine whether an object contains all keywords in $q.W$, we pre-compute and maintain a global inverted list that maps each word to the objects containing such word, denoted by $\mathcal{I}(w)$. Thus, we can efficiently obtain the promising objects containing the $q.W$ by computing the intersection of $\mathcal{I}(w_i)$.



(a) Rare word "palace". (b) Frequent word "park".

Figure 8: Two circumstances for occurrence list.

Figure 7 provides an example to illustrate the method. Consider a query with $W = \{\texttt{park}\}$ and prefix "s". We first obtain the object list using inverted list $\mathcal{I}(\texttt{park})$, i.e., $\{o_4, o_8, o_9\}$. Then, we examine each object contained in a leaf node, say "st", and only enqueue the objects contained in $\mathcal{I}(\texttt{park})$, i.e., o_4.

Next, we analyse the time complexity of the above method. Since the size of `ObjectList` in a leaf is strictly less than or equal to M, and the size of the $\mathcal{I}(w)$ is maximally equal to the size of the data N, thus the time complexity for intersection on one node is $\mathcal{O}(M \log N \cdot |W|)$.

However, in the worst case, we have to examine all the leaf nodes, which leads to low performance. Thus, according to Lemma 1, the overall time complexity is $\mathcal{O}(NM \log N \cdot |W|)$. Obviously, this method visits many *unnecessary* leaf nodes which do not contain all keywords in $q.W$. For example, given query "`palace s`", for the prefix "`s`", there may be many leaf nodes that need to be visited. However, the number of leaf nodes containing word "`palace`" is rather small (See Figure 8(a)). We can see that the expanding child nodes labelled with '2', '3' is useless for producing the answers.

4.2.2 A Cost-based Method

To avoid unnecessary traversal, we must include additional textual information for early termination. Inspired by our observation in Figure 8(a), for any keyword w on node n, we can explicitly use a child list to indicate that only the children of n in the list will contain the keyword w. We call such a list an *occurrence list*, denoted by $\texttt{OL}(n, w)$. Thus, when traversing to such a node n, we only need to expand $\cap\ \texttt{OL}(n, w_i), w_i \in q.W$ into the priority queue to prune the unpromising descendants.

Obviously, if we store occurrence lists for all possible keywords on every tree node, the pruning power will be fully utilized. However, it will result in high memory cost, since the space requirement for storage will be $N \cdot |Vocabulary|$ in a worst case. In fact, sometimes occurrence list may have no effect on the performance. It happens when a keyword is frequent in the spatial database, and thus many leaf nodes may contain such word. Figure 8(b) shows this case. The keyword "park" appears in the sub-tree of every child of the root node, and thus it is useless to store occurrence list for "`park`" on the root node.

Therefore, we need to judiciously select keywords and nodes for storing occurrence lists. To this end, we first consider the *cost* of storing $\texttt{OL}(n, w)$ as its length.

DEFINITION 3 (COST).

$$Cost(\texttt{OL}(n, w)) = Length(\texttt{OL}(n, w))$$

Each $\texttt{OL}(n, w)$ will benefit query processing by ignoring those child nodes of n which do not contain keyword w. Therefore, the benefit can be described in two parts: (1)

the overhead saved on priority queue operations, e.g. enqueue or dequeue, and (2) the overhead saved on leaf nodes intersection. Let $\texttt{CL}(n)$ denote the original children list for node n. Let $\texttt{PQO}(n')$ denote the overhead on priority queue operations for all nodes rooted at n' and $\texttt{LNO}(n')$ denote the overhead on intersection for all leaf nodes in sub-tree of n', we define the benefit of $\texttt{OL}(n,w)$ as:

DEFINITION 4 (BENEFIT).

$$Benefit(\texttt{OL}(n,w)) = \Sigma_{\{n'|n'\in\texttt{CL}(n)\wedge n'\notin\texttt{OL}(n,w)\}}(\texttt{PQO}(n') + \texttt{LNO}(n')),$$

where $p(w)$ is the probability for word w being queried.

Conventionally, $\texttt{PQO}(n')$ is equal to the number of tree nodes under n', and $\texttt{PQO}(n')$ is estimated by the number of objects under n'.

Finally, for a given amount of memory budget $\mathcal{B}$, we want to maximize the overall benefit for different selected keywords and nodes. The formal problem is stated as follows.

DEFINITION 5. (MEMORY-CONSTRAINED KEYWORDS NODES SELECTION OPTIMIZATION) *Given the budget $\mathcal{B}$, the keyword set $\mathcal{K}$ and the node set $\mathcal{N}$, we want to find a collection of* keyword-node *pairs, that achieve maximum benefits within the budget $\mathcal{B}$.*

$$(n^*, w^*) \leftarrow \underset{\forall(n,w)\in\mathcal{K}\times\mathcal{N}}{\text{argmax}} Benefit(\texttt{OL}(n,w))$$
$$s.t. \sum Cost(\texttt{OL}(n,w)) \leq \mathcal{B}\ .$$

We now prove this problem is NP-Complete.

THEOREM 1. *Memory-Constrained Keywords and Nodes Selection Optimization is NP-Complete.*

PROOF. The problem can be reduced from the 0-1 Knapsack Problem. Since the budget $\mathcal{B}$ is equivalent to the knapsack's volume, $Cost(\texttt{OL}(n,w))$ is the size of the items and $Benefit(\texttt{OL}(n,w))$ can be looked as item's value. Thus, proof is done. □

We propose a greedy heuristic algorithm to solve this problem. We sort all the (n,w) by $\frac{Benefit(\texttt{OL}(n,w))}{Cost(\texttt{OL}(n,w)}$ in descending order, then choose the current best (n,w) so long as the budget is not exhausted. Previous study [12] showed that the approximate ratio for greedy algorithm is 2. Note that, the $Cost(\texttt{OL}(n,w)$ is far smaller than the budget $\mathcal{B}$, thus the results would not be bad. Example 2 shows a scenario for multi-keyword search on a materialized PR-Tree.

EXAMPLE 2. *Consider a multi-keyword query "*`palace s`*" at location (40.5,-74.0).*
Step 1*: We start to push the ⟨∅,0⟩ into the queue:*
Queue: ⟨∅,0⟩
Step 2*: ⟨∅,0⟩ is dequeued, and we check child nodes ⟨s,1.84⟩, ⟨s,1.22⟩ and ⟨s,1.38⟩. Since only the node ⟨s,1.84⟩ has been materialized with keyword "*`palace`*", we discard the other two unpromising nodes and only enqueue node ⟨s,1.84⟩.*
Queue: ⟨s,1.84⟩
Step 3*: The objects list for node ⟨s,1.84⟩ is {2,4,13}, and we intersect it with the global inverted list for keywords "*`palace`*" {2}. Finally only object o_2 has been enqueued;*
Queue: ⟨o_2,2.76⟩
Thus, we get the answers for query "`palace s`*" on the materialized PR-Tree.*

5. EXTENSION ON RANKING FUNCTIONS

Existing studies on keyword search in spatial database [5] can support more sophisticated ranking functions, which consider not only the spatial proximity but also textual relevancy between queries and objects. Here we discuss how to use our PR-Tree to support such ranking functions.

Given a query q and an object o, a general ranking function D_{st} to compute their similarity can be defined as

$$D_{st}(q,o) \quad = \quad \alpha(1 - \frac{\texttt{dist}(q,o)}{maxD_s}) + (1-\alpha)\frac{\texttt{rel}(q,o)}{maxD_t},$$

where $maxD_s$ is the maximum distance on space, $maxD_t$ is the maximum distance on text, and $\texttt{rel}(q,o)$ is the tf*idf based textual relevance between q and o.

$$\texttt{rel}(q,o) = \sum_{s\in q.W} tf(s,o) * idf(s,\mathcal{O}),$$

where $tf(w,o)$ is the term frequency of term w in object o and $idf(w,\mathcal{O})$ is the inverse document frequency of term w in the object set $\mathcal{O}$.

Next we discuss how to use the PR-Tree to support the new ranking function. We add a list of $\langle keyword, weight\rangle$ pairs on each node of the PR-Tree, which indicates that for all objects in the sub-tree of this node, the maximum tf*idf for *keyword* is *weight*. Formally speaking, on a tree node f and for a specific keyword w, the *weight* is calculated by:

$$weight(w,f) = \max_{o\in\texttt{SubTreeObj}(f)} tf(w,f) * idf(w,\mathcal{O}).$$

where $\texttt{SubTreeObj}(f)$ denotes all the objects in the sub-tree of node f.

At query time, we can compute the textual relevancy by $\sum_{w\in q.W} weight(w,f)$, then aggregate the spatial distance to obtain the ranking score $D_{st}(q,f)$. This overall value can be used as the maximum boundary value for a given query q at node f on the PR-Tree, facilitating effective pruning.

Similarly, our PR-Tree is also capable of supporting the ranking function in [21].

6. EXPERIMENT

In this section, we report the experimental results. We implemented two baseline algorithms, FEH [13] and MT [21], as mentioned in Section 2.2. For FEH, we used R*-tree [2] as reported in the paper. We compared our PR-Tree based method (denoted by PRT) with the two baseline methods.

6.1 Experiment Setup

We used two real datasets in our experiments.

1) OPENSTREET**:** We extracted 2 million POIs for day-life in USA, such as government buildings, parking lots, schools, etc., from the OpenStreet open-source spatial database[5].

2) BUSINESS**:** We obtained 5 million POIs for business in USA, such as cafe, company, restaurant, etc., from a popular directory website, Factual[6].

Table 6.1 provides some statistics of the two data sets, such as number of objects, data size, etc.

All baseline indices and our PR-Tree were memory-resident. The parameters of the baseline algorithms were set as the default values in the original papers. All the algorithms were implemented in C++, and all the experiments were

[5]http://planet.openstreetmap.org
[6]http://www.factual.com

Table 1: Statistics of the two datasets.

Dataset	OpenStreet	Business
Number of objects	2,003,608	5,073,369
Original Data Size	183M	1,461M
# of distinct words	97,342	565,510
Avg. # of words per object	3.23	5.45

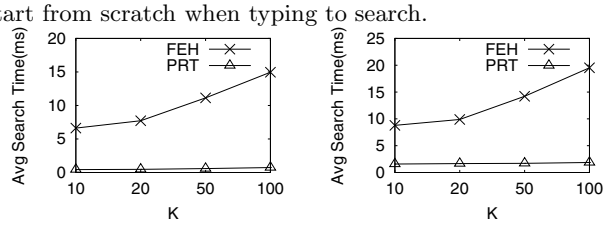
conducted on an 2.66MHz Intel Xeon processor and 24GB RAM, running on Ubuntu 11.04.

6.2 Comparison for Single-Prefix Queries

We evaluated the query performance for single prefix queries. Since the MT algorithm could not support word segmentation on text, we conducted the following two experiments. We first compared our method PRT with FEH under the word-segmentation setting, and then compared PRT with MT by taking the whole text of each object as a single string.

6.2.1 PRT vs. FEH

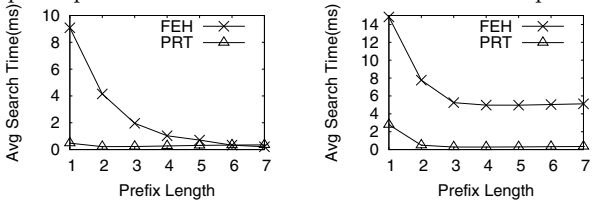
We randomly chose 1000 words from the vocabulary, and generated three prefixes from one selected word. For example, consider a selected word "`park`". We generated three prefixes, i.e., "`p`", "`pa`" and "`par`" for query evaluation. Compared with long prefixes, the short prefixes were much better for evaluating the query performance, since users always start from scratch when typing to search.



(a) OpenStreet dataset. (b) Business dataset.

Figure 9: PRT vs. FEH for single-prefix queries with different k (prefix lengths are within $[1, 3]$).

Figure 9 shows the experimental results on the two datasets with different k. We can see that our algorithm outperformed the FEH algorithm in both datasets. On the OpenStreet dataset, our PRT algorithm is about 15 times faster than FEH. On the Business dataset, PRT was about 5 times faster than FEH. The improvement of the performance was mainly attributed to our novel index structure, since PR-Tree can utilize both textual and spatial pruning simultaneously. Using the PR-Tree, we can efficiently find the promising objects which not only contain the query prefix, but also are near our query location. In contrast, FEH only utilizes a pure spatial index and does not consider textual pruning.



(a) OpenStreet dataset. (b) Business dataset.

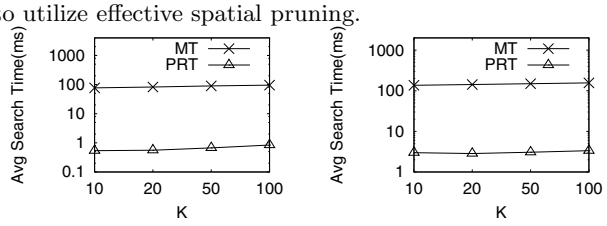
Figure 10: PRT vs. FEH for single-prefix queries with different prefix lengths ($k = 20$).

We also conducted an experiment to evaluate the performance on different prefix lengths between PRT and FEH. Given 1,000 selected words mentioned above, we generated all possible prefixes for each word. Figure 10 provides the results for $k = 20$. Though there were lots of candidates for shorter prefixes, they could still be distinguished by different nodes on PR-Tree that represent different regions. Thus, PRT achieved better performance than FEH.

6.2.2 PRT vs. MT

To compare PRT and MT, we took the text of each POI as a single string without word-segmentation. Similar to previous experiments, we generated 1000 prefixes whose length were within $[1, 3]$. Figure 11 provides the experimental results on both datasets for different k values.

We can see that the PRT algorithm was much more efficient than the MT algorithm. The poor performance of MT was due to its trie-based index structure, which failed to utilize effective spatial pruning.



(a) OpenStreet dataset. (b) Business dataset.

Figure 11: PRT vs. MT for single-prefix queries with different k (prefix lengths are within $[1, 3]$).

6.2.3 Index Sizes

We then examined the space complexity of different indices. Table 2 provides the memory usage on two data sets. As shown, PRT consumed less memory, compared with FEH and MT. To achieve better performance, FEH maintained two filters on each R-Tree node for a huge amount of prefixes, and MT materialized a lot of tree nodes with a small granularity of regions. Compared with these algorithms, our method only maintained a PR-Tree, which scaled to the size of original data as proved in Section 3.5.2.

Table 2: Index size for three algorithms (MB).

	Data Size	PRT	FEH	MT
OpenStreet	183	122	223	231
Business	1,461	487	1,328	2,344

6.3 Comparison for Multi-Keyword Queries

6.3.1 Tuning of Materialization Size

We examined the query performance of our cost based algorithm (Section 4.2) by varying memory budget $\mathcal{B}$, to investigate the impact of materialization size on the performance. On each data set, we randomly selected 1000 objects and randomly chose 3 keywords from $o.W$ in each selected objects as a query. Then, we took the first two keywords as complete words, and generated a random length of prefix from the last keyword. For example, given an object with text "`Washington State Driving School`", one possible keyword query generated from it was "`Driving Washington Sc`".

Figure 12 provides the experimental results for multi-keyword queries on the OpenStreet dataset. We can see that different materialization sizes had great impact on query performance. Specifically, we can see that the average query time

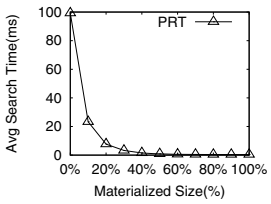


Figure 12: Performance with different budgets.

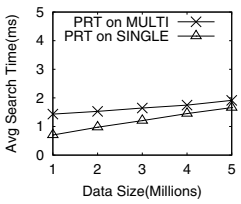


Figure 13: Time Scalability.

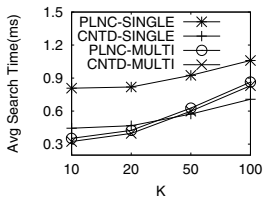


Figure 14: On two spatial partition strategies

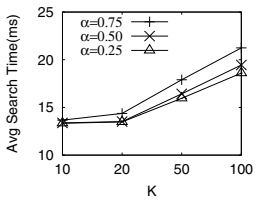
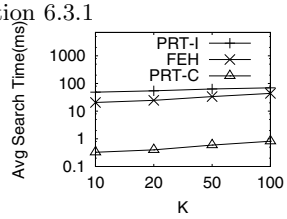


Figure 15: On tf*idf ranking function.

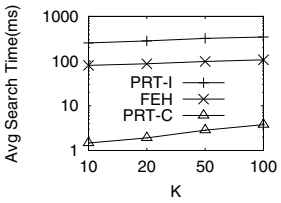
varies dramatically when $\mathcal{B}$ is 0% to 60% of the original data size. When $\mathcal{B}$ is larger, the difference is not significant. Based on this observation, we could judiciously select a proper memory budget in practice, in order to balance the space and time. In the remainder of our experiments, we set the budget percentage as 70%.

6.3.2 Efficiency on Multi-Keyword Search

We evaluated performance for multi-keywords queries for our intersection-based method (denoted by PRT-I), cost-based method (denoted by PRT-C) and the baseline FEH algorithm. All the settings were the same as those in Section 6.3.1



(a) OpenStreet dataset.



(b) On Business dataset.

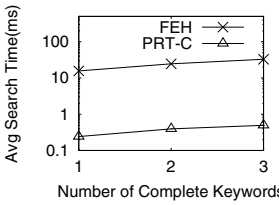
Figure 16: PRT vs. FEH for multi-keyword queries with different k (2 complete keyword and 1 prefix).

Figure 16 provides the experimental results on the two datasets for different k values. We can see that our PRT-C algorithm was approximately 100 times faster than the FEH algorithm for both data set. This significant improvement results from the fact that PR-Tree has guaranteed that the visited leaf nodes satisfy the constraints on query prefix $q.p$, thus it would not bring any unnecessary work of traversal. On the other hand, FEH might traverse many unpromising leaf nodes which could not produce any top-k answer. Moreover, the PRT-I algorithm achieved poor performance. This is because that PRT-I did not utilize the occurrence list and a lot of unnecessary nodes might be visited at query time.

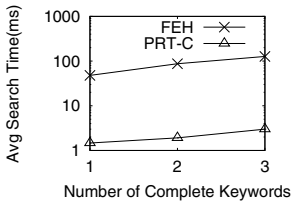
In addition, we compare the algorithms by varying different numbers of complete keywords. Figure 17 provides the experimental results. As PRT-C was always better than PRT-I, in the figure we only show the results of PRT-C and FEH. We can see that the time of PRT increases sub-linearly with the increase of complete keyword numbers. This shows that our cost-based method of materialization achieves good performance on multi-keywords queries.

6.4 On Spatial Partitioning Strategies

We evaluated the query performance on two spatial partitioning settings, i.e., planar center or centroid as the quad-split point, which mentioned in Section 3.3. We first built two different PR-Trees, then applied single prefix and multi-keyword queries to make comparisons. All the query settings were the same as those in Sections 6.2 and 6.3.



(a) OpenStreet dataset.



(b) Business dataset.

Figure 17: PRT vs. FEH for multi-keyword queries with different keyword numbers.

Figure 14 shows the results, where "PLNC" denotes the "planar center" and "CTND" denotes the "centroid". We can see that, by using centroid as the splitting point, we could achieve about 40% speed improvement for single prefix query compared to planar center. This is because using the centroid will result in a more even partitioning of objects on space. However, for multi-keyword queries, the improvement is not significant. The reason is that utilization of occurrence list results in early terminations, and thus long branches have little effect on the query time.

6.5 Scalability

We examined the scalability of our best algorithm on the Business dataset, since it contains POIs in the whole region of USA. We used the same experimental settings as we applied in Sections 6.2 and 6.3.

6.5.1 Time Scalability

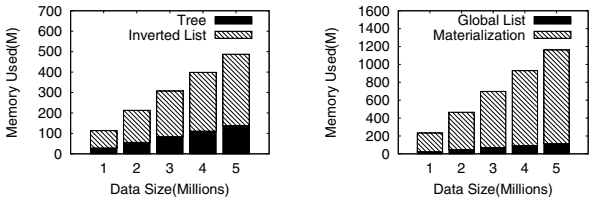
We first evaluated the time scalability for single-prefix and multi-keyword queries with different sizes of the original data. Figures 13 shows the experimental results. We can see that the query time for both queries increased sub-linearly when the data size increased from 1 to 5 millions. From the results we can see that our PRT algorithm could support location-aware instant search on large datasets.

6.5.2 Space Scalability

We evaluated the space scalability of PRT. Figure 18(a) provides the experimental results for single-prefix queries. We can see that with the increase of data sizes, both the size of the PR-Tree and the inverted lists in the leaf nodes increased sub-linearly, which was consistent with the analysis on space complexity mentioned in Section 3.5.2. To further improve the storage utility, we can store the inverted list in disk, and only maintain the PR-Tree in memory.

Figure 18(b) provides the experimental results for multi-keyword queries, where the global keyword inverted index and keyword occurrence materialization were included. We can see that the size of the global keyword inverted index consumed little memory, and the total materialization size

was adjusted linearly to the size of the data. Thus the indexing memory of our PR-tree also scaled to the data size.



(a) Single-prefix queries. (b) Multi-keyword queries.

Figure 18: Space Scalability.

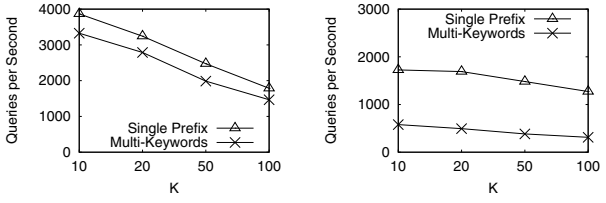
6.6 Evaluation on Construction and Updates

In this section, we evaluated the performance of PR-Tree operations, i.e., construction, insertion, deletion, on the two datasets. We first conducted 1000 insertion and deletion operations for testing. Figure 3 shows that the PR-Tree can be efficiently maintained.

Table 3: Performance of tree operations.

Operation	OpenStreet	Business
Construction	147 s	1066 s
Materialization	432 s	6559 s
Insertion per Object	0.480 ms	2.940 ms
Deletion per Object	0.093 ms	0.290 ms

We also conducted the throughput experiments on PRT. We generated a series of operations, i.e., 10% insertions, 10% deletions and 80% random queries. Figure 19 shows the high throughput of our PRT algorithm.



(a) OpenStreet dataset. (b) Business dataset.

Figure 19: Evaluation on throughput (10% insertions, 10% deletions, 80% top-k queries).

6.7 Evaluating Ranking Functions

We evaluated the efficiency of the *tf*idf* based ranking function mentioned in Section 5. We generated 1000 queries, each of which only contained complete words(the length varied from 1 to 3).

Figure 15 provides the experimental results on the OpenStreet dataset with different k values. We can see that with the increase of parameter α, the average query time decreased. This is because for large parameter α, the impact of spatial filtering became insignificant. The experimental results showed that although we focus on prefix search, the PR-Tree can still support the *tf*idf* ranking function.

7. CONCLUSIONS

In this paper, we have studied the problem of location-aware instant search. We proposed a novel indexing structure PR-Tree to support location-aware instant search. We used prefix-regions to partition the spatial data by considering the spatial information and textual description simultaneously. We discussed how to construct and update the PR-Tree. Using the PR-Tree, we devised effective algorithms to support single prefix queries and multi-keyword queries. We proposed a best-first traversal based algorithm to answer single prefix queries. We developed an intersection-based algorithm and a cost-based algorithm to answer multi-keyword queries. We have implemented our techniques and experimental results show that our method achieved high performance and outperformed state-of-the-art approaches.

8. ACKNOWLEDGEMENT

This work was partly supported by the National Natural Science Foundation of China under Grant No. 61003004, the National Grand Fundamental Research 973 Program of China under Grant No. 2011CB302206, a project of Tsinghua University under Grant No. 20111081073, and the "NExT Research Center" funded by MDA, Singapore, under the Grant No. WBS:R-252-300-001-490.

9. REFERENCES

[1] H. Bast and I. Weber. Type less, find more: fast autocompletion search with a succinct index. In *SIGIR*, pages 364–371, 2006.
[2] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD Conference*, pages 322–331, 1990.
[3] S. Chaudhuri and R. Kaushik. Extending autocompletion to tolerate errors. In *SIGMOD Conference*, pages 707–718, 2009.
[4] Y.-Y. Chen, T. Suel, and A. Markowetz. Efficient query processing in geographic web search engines. In *SIGMOD Conference*, pages 277–288, 2006.
[5] G. Cong, C. S. Jensen, and D. Wu. Efficient retrieval of the top-k most relevant spatial web objects. *PVLDB*, 2009.
[6] J. Fan, G. Li, L. Zhou, S. Chen, and J. Hu. Seal: Spatio-textual similarity search. *PVLDB*, 5(9):824–835, 2012.
[7] I. D. Felipe, V. Hristidis, and N. Rishe. Keyword search on spatial databases. In *ICDE*, 2008.
[8] R. A. Finkel and J. L. Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Inf.*, 4:1–9, 1974.
[9] R. Hariharan, B. Hore, C. Li, and S. Mehrotra. Processing spatial-keyword (SK) queries in geographic information retrieval (GIR) systems. In *SSDBM*, 2007.
[10] G. R. Hjaltason and H. Samet. Ranking in spatial databases. In *SSD*, pages 83–95, 1995.
[11] W. Huang, G. Li, K.-L. Tan, and J. Feng. Efficient safe-region construction for moving top-k spatial keyword queries. In *CIKM*, 2012.
[12] O. H. Ibarra and C. E. Kim. Fast approximation algorithms for the knapsack and sum of subset problems. *J. ACM*, 22(4):463–468, 1975.
[13] S. Ji and C. Li. Location-based instant search. In *SSDBM*, pages 17–36, 2011.
[14] S. Ji, G. Li, C. Li, and J. Feng. Efficient interactive fuzzy keyword search. In *WWW*, pages 371–380, 2009.
[15] G. Li, J. Feng, and C. Li. Supporting search-as-you-type using sql in databases. *IEEE TKDE*, 2011.
[16] G. Li, S. Ji, C. Li, and J. Feng. Efficient type-ahead search on relational data: a tastier approach. In *SIGMOD Conference*, pages 695–706, 2009.
[17] G. Li, S. Ji, C. Li, and J. Feng. Efficient fuzzy full-text type-ahead search. *VLDB J.*, 20(4):617–640, 2011.
[18] G. Li, J. Wang, C. Li, and J. Feng. Supporting efficient top-k queries in fuzzy type-ahead search. In *SIGIR*, 2012.
[19] G. Li, J. Xu, and J. Feng. Desks: Direction-aware spatial keyword search. In *ICDE*, 2012.
[20] J. Lu, Y. Lu, and G. Cong. Reverse spatial and textual k nearest neighbor search. In *SIGMOD Conference*, pages 349–360, 2011.
[21] S. B. Roy and K. Chakrabarti. Location-aware type ahead search on spatial databases: semantics and efficiency. In *SIGMOD Conference*, pages 361–372, 2011.
[22] D. Wu, M. L. Yiu, C. S. Jensen, and G. Cong. Efficient continuously moving top-k spatial keyword query processing. In *ICDE*, pages 541–552, 2011.
[23] B. Yao, F. Li, M. Hadjieleftheriou, and K. Hou. Approximate string search in spatial databases. In *ICDE*, 2010.
[24] Y. Zhou, X. Xie, C. Wang, Y. Gong, and W.-Y. Ma. Hybrid index structures for location-based web search. In *CIKM*, 2005.