# Loquat: An Interactive System Design for Location-Aware Query Autocompletion

Sheng Hu, Chuan Xiao, and Yoshiharu Ishikawa

***Abstract*—Query autocompletion (QAC) is the feature to provide the intended possible candidate completions given some initial prefixes from users. By applying QAC techniques, users are assisted in formulating queries and saving input keystrokes. Due to the convenience it brings to users, QAC has been adopted in many real-world applications, including search engines, integrated development environments (IDEs), and mobile devices. With the growing popularity of mobile devices, a recent trend is to integrate query autocompletion into location-based services, such as Web mapping and spatial keyword search. In this paper, we present an interactive system of location-aware query autocompletion called Loquat, which provides a graphical interface to help users easily formulate their location-aware queries. We develop novel index structures and search algorithms to make such an interactive system work efficiently. We extend our system to support fuzzy search and multiple-keyword search. We also define a new ranking function taking fuzzy threshold value into consideration. The experiments on two real-life datasets verify the efficiency and its interactive usability of our system.**

***Index Terms*—Query autocompletion, spatial databases, interactive.**

## I. Introduction

The prevalence of geo-tagged text data in modern real-world applications such as Web mapping and spatial-keyword search has led to a rejuvenation of research on geo-text data managements. However, typing meaningful spatial keyword query is a tedious and error-prone process especially on devices with small keyboards such as mobile phones. Query autocompletion plays an important role in search engines, command shells, desktop search, software development environments, and mobile applications. It can guide users to create queries faster and avoid spelling errors by providing instant completions to users' query letter by letter. It can also improve the throughput of the system as query or intermediate results can be effectively cached and reused. With the growing popularity of mobile devices, a recent trend is to integrate query autocompletion into location-based services. One of the main applications is to complete the queries with the textual descriptions of nearby points of interest within distances bounded by a threshold value, such as a Web mapping service illustrated in Fig. 1. We call this problem *location-aware query autocompletion*. A query of this problem includes a location, such as the **black arrow mark** in Fig. 1, which can be obtained automatically by tracking the GPS signal of the mobile devices. The query also includes a string prefix, a point of interest will be returned if it is close to the spatial location and its textual descriptions begin with the given string prefix. Moreover, fuzzy search, or error-tolerant autocompletion also becomes very prevalent, especially considering the case that users might type with the error-prone keyboards of mobile devices. With the help of fuzzy search features, the system can suggest correct results despite there are typos on both query prefix and data sides. To handle the location-aware query autocompletion problem, existing methods focus on combining spatial and textual information to process queries efficiently. A comprehensive prospective of these methods can be presented in a taxonomy, according to how their indexes are combined. We classify them into text-first [1], space-first [2], and tightly-combined [3] methods. For text-first methods, a trie is used to index string descriptions of data objects. Meanwhile, objects and the information of the locations can be retrieved on leaf nodes of the trie. For space-first methods, an R-tree or quadtree is used to index data objects by their locations, and use the textual descriptions as filters when processing queries. For tightly-combined methods, integrated descriptors of both textual and spatial information are designed to build the index. However, all of the existing methods have the drawbacks of low runtime performance when come across the scalable data, and the response time latency becomes unendurable especially when large amount of simultaneous queries occur. In addition, these methods also ignore to support fuzzy search features which might be an even worse case for system workload. To avoid the problem brought in by storing on trie nodes the spatial information of all the queries, we choose to only store the spatial information of data objects instead.

In this paper, we show that by employing novel index structures and algorithms, high speed for interactive and fuzzy search performance can be achieved. We implemented these algorithms and techniques in a system called Loquat (**Lo**cation-aware **qu**ery **a**u**t**ocompletion). A part of the preliminary algorithmic aspect of this work was presented previously in a journal [4]. We extend our system to support fuzzy search and multiple-keyword search. We also define a new ranking function taking fuzzy threshold value into consideration. Finally, we demonstrate the superiority of our system compared with the state-of-the-art work through the experiments.

Manuscript received August 10, 2018; revised November 12, 2108. This research was partly supported by the Grant-in-Aid for Scientific Research (16H01722) from JSPS.

Sheng Hu and Ishikawa Yoshiharu are with Graduate School of Informatics, Nagoya University, Nagoya, 464-0804 Japan (e-mail: hu@db.ss.is.nagoya-u.ac.jp, ishikawa@i.nagoya-u.ac.jp).

Chuan Xiao is with Institute for Advanced Research, Nagoya University, Nagoya, 464-0804 Japan (e-mail: chuanx@nagoya-u.jp).

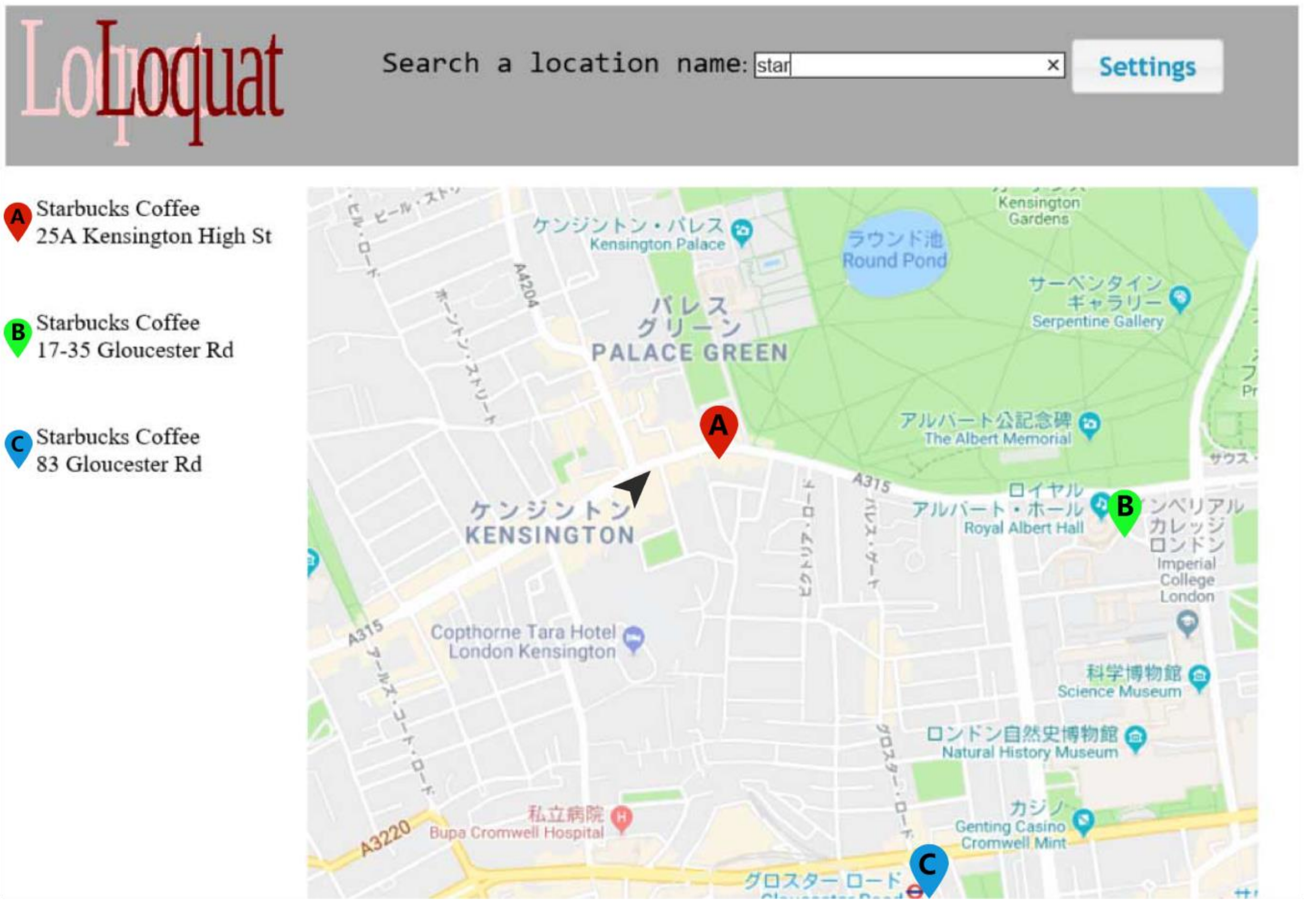

Fig. 1. Interface of loquat system.

Fig. 1 is a screenshot of our system when a user tried finding POIs begin with “star”. The user intended to find the nearest Starbucks coffee for a rest and wanted to find the one with the highest rating and providing good services. The user’s location was obtained by GPS signal and marked on the map as a black arrow. The search results returned on the left side were refreshed incrementally when the query was typed letter by letter. Finally, when the user stopped at “star”, three qualified POIs were returned as A, B and C. A was ranked highest because it is nearest and also has a highest rating.

The Loquat system provides several customized features to make it user-friendly. By changing the search settings by clicking the setting button, users can choose to select a rectangle area on the map to return all qualified results instead of return the highest ranked k ones. Also, users can switch the fuzzy search option to decide to perform an exact match or not. Users are also allowed to specify how many typos can be tolerant as errors. Moreover, user can set a weight between distance and POI popularity (ratings, price, etc.).

Our contributions can be summarized as follows:

• We implement a highly interactive and efficient location-aware autocompletion system called Loquat. It has a user-friendly interface and a lot of user-customized settings. It can efficiently answer range and top-*k* queries with an acceptable index size. Both range queries and top-*k* queries can be answered in microseconds or even faster.

• We extend the system to support fuzzy search and multiple-keyword search so as to handle the case when users input queries with error-prone devices.

• We conduct experiments to evaluate the efficiency and interactive usability of our system with comparison to the state-of-the-art work. The rest of this paper is organized as follows. Section II is an overview of Loquat. Section III presents more technical details including index structure and query processing algorithms. Section IV reports experiment results and analysis. Section V surveys related work. Section VI concludes this paper.

## II. Overview of Loquat

### A. System Architecture

The overall architecture of Loquat is shown in Fig. 2. The system adopts client/server architecture. The client-side is based on a web-based interface and can be accessed by mobile phones, desktop PC, tablets and so on. The user inputs his query through the interactive user interface, and the web interface acts like a client to transmit the formatted query to our web server. The server-side is composed of (1) a Query Processing module, (2) an Index module, (3) a Results Ranking module and (4) a Geo-Textual Database module. The Query Processing module receives a formatted query from the user client and transforms the raw query into the form which will be executed directly in the Index module. The index module indexes the textual data as a prefix-tree (trie) with additional spatial information for fast look-ups. After executing the query passed from Query Processing module, the Index module returns all possible candidate results to Result Ranking module. Then, the Results Ranking module ranks the results according to some specific criterions and returns the top-$k$ results to the user client. The user interface will receive the top-$k$ results and show the results in a drop-down list in an incremental way. The Geo-Textual Database module will collect the POIs information regularly and update the trie in the Index module incrementally. Except for the Geo-Textual database module, the other three modules will be run in memory.

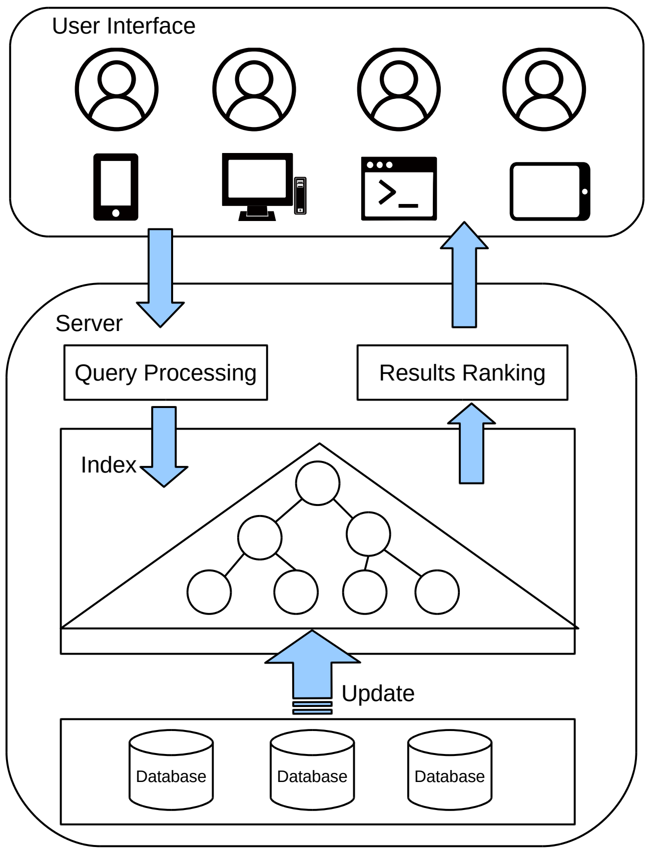

Fig. 2. The system architecture of loquat.

### B. Problem Formulation

Consider a geo-textual database $O$. Each object $o \in O$ is defined as a tuple $\{o.str, o.loc, o.scr\}$, where $o.str$ is the text which describes the objects. $o.loc = (x, y)$ is a descriptor and describes the location in a 2-dimensional space.

| Object ID | *o.str* | *o.loc* | *o.scr* |
|---|---|---|---|
| $o_1$ | navitime | (24, 25) | 0.4 |
| $o_2$ | nagoyadome | (18, 12) | 0.9 |
| $o_3$ | nagoyaport | (11, 19) | 0.8 |
| $o_4$ | nursing | (1, 19) | 0.7 |
| $o_5$ | stone | (7, 27) | 0.1 |
| $o_6$ | studio | (27, 12) | 0.1 |
| $o_7$ | starbucks | (22, 18) | 1.0 |
| $o_8$ | starboost | (5, 5) | 0.3 |
| $o_9$ | station | (19, 9) | 0.8 |
| $o_{10}$ | school | (15, 29) | 0.6 |

Fig. 3. An example of database $O$.

$o.scr$ is the static score aggregated from several features, e.g. user rating and price. $max\_scr$ is the maximum static score of the objects. $max\_dist$ is the maximum distance between two objects in $O$. An example is shown in Fig. 3 and Fig. 4.

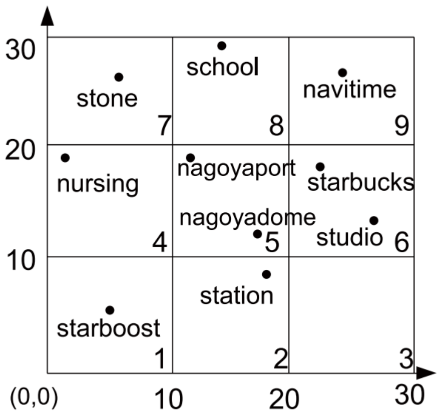

Fig. 4. $O$ in 2-dimensional space.

Consider two strings $s$ and $s'$, "$s' \prec s$" denotes that $s'$ is a prefix of $s$; i.e., $s' = s[1..|s'|]$. We introduce two types of queries as below:

**Range Query.** The query $q$ is composed of a query string $q.str$ which is the input prefix and a range $q.rng$ depicted by a rectangle. The results to the query $q$ is a set of the objects $o \in O$ such that $q.str \prec o.str$ and $o.loc$ is in the range $q.rng$.

**Top-*k* Query.** The query $q$ is composed of a query string $q.str$ which is the input prefix and a location $q.loc$. The results to the query $q$ is a ranked set of the top-$k$ objects $o \in O$ such that $q.str \prec o.str$, ranked by a ranking function $F(o, q)$. A following ranking function is defined to combine some normalized factors of an object with regarding to the query $q$, In addition, our method can be extended to support other monotonic functions.

$$F(o,q) = \alpha \frac{o.scr}{max\_scr} + (1-\alpha)\left(1 - \frac{dist(o.loc, q.loc)}{max\_dist}\right) \quad (1)$$

where α is a weight parameter to balance spatial proximity and the static score. $\frac{o.scr}{max\_scr}$ is the normalized score within the range of [0, 1] which measures the popularity with the object. $\left(1 - \frac{dist(o.loc, q.loc)}{max\_dist}\right)$ is the normalized Euclidean distance between the object and the query.

## III. Technical Details

### A. Index Structure

We build our index as a trie on the set of object strings. Each string corresponds to the labeled path from the root to a node in the trie, then the traversal can be quickly done by starting from the root node to locate by going along the path matched by the query string. Next, we give a definition of the *underlying object*. If a data object appears as a result of a trie node, that is, the path from the root to the trie node is a prefix of the data object, then we call a data object an underlying object of this trie node. Note that a trie node may have many underlying objects. For easy illustration, we equip each trie node with a unique id by running a pre-order traversal in the trie as shown in Fig. 5. Meanwhile, for each trie node, we integrate spatial information into it. First, the global space is partitioned into a set of spatial regions. This step can be done using common data structures for spatial objects, such as grid, R-tree, quadtree, etc. As shown in Fig. 4, the global space is partitioned into nine regions. These regions can be obtained by the cells of grids or the leaf nodes of a quadtree. For ease of illustration, we use a grid to partition the global space in Fig. 4, then denote each region with a region number in the left-down corner of that region. Note that in our experiments, quadtree is used because of better practical performance. According to the number order attached in each region, a bit array is designed with each position of the array representing a region. We equip each node in the trie with such a bit array. We set the corresponding bit to 1 if the node has an underlying object in this region; or 0, otherwise. Especially, we call it *region bit array*. By the intersection operation of this bit array, we can check whether there is an underlying object in the region by simply intersecting it with the query range.

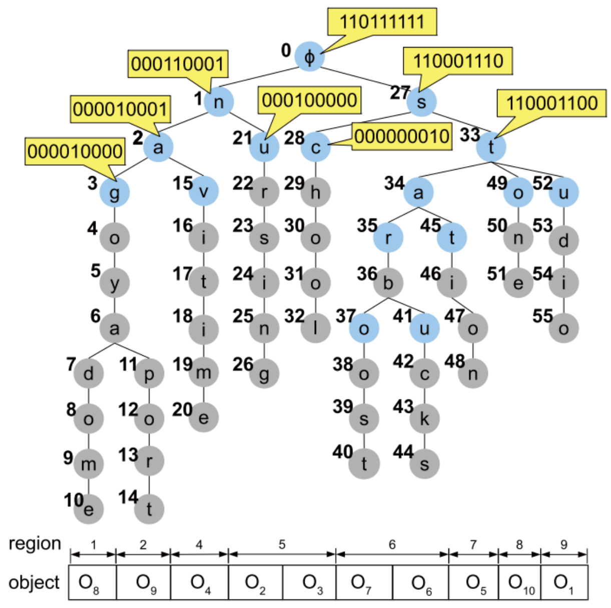

Fig. 5. The bit trie index.

Next we introduce the data objects storage structure. We store all the data objects into an array called *data object array*, which is partitioned similarly into spatial regions according to the partitions on the global space. Then we begin to sort the data object array by two kinds of orders. First we sort them as the region order shown in Fig. 5. Then we sort the partial array corresponding to each region according to the order of the leaf node appearing in the trie. Next, we equipped each trie node with a list called *region list*, whose entries are in the form of < region ID, maximum static score, starting pointer, ending pointer >, to efficiently locate the underlying objects in the array. The maximum static score of an entry is computed in advance as the maximum static score among all the underlying objects of that node in this region, and it is used for early termination when answering top-*k* queries. The starting and ending pointers are used to quickly locate results in the data object array with a linear scan. They can be recorded as the index of the starting and ending positions in the partial array which contains the underlying objects of the node in this region, respectively. For the purpose of early termination, we sort the entries in the list by descending maximum static score order.

For index construction, we first sort the data strings in alphabetical order, then insert them into the trie one by one. When a string is inserted, we update the region bit arrays and the region lists of the nodes on the path, along with the maximum static scores of the nodes. The time complexity of the index construction is $O(O\log O + S)$, where $S$ is the sum of string lengths of the objects. Consider the strings in Fig. 3, its trie is shown in Fig. 5. For node 2, its corresponding prefix is **na**, its bit array is “000010001”, and its underlying objects are $o_1$, $o_2$ and $o_3$.

## B. Search Algorithm

We introduce several query processing algorithms in this section. First, range queries and top-k queries algorithms are shown. Then, we extend them to support fuzzy search and discuss the way to support multiple-keyword search.

**Algorithm 1:** RangeQuerySearch $(q, T)$

```
b ← InitRegionStatus($q.rng$);
n ← the root of T;
foreach keystroke $q.str[i]$ do
    if n has a child $n'$ through label $q.str[i]$ then
        b ← b AND the region bit array of $n'$;
        if $b \neq 0$ then
            $n \leftarrow n'$;
        else
            n ← null;
            break;
    else
        n ← null;
        break;
return n, b
```

**Algorithm 2:** RangeQueryFetchResult $(q, n, b, T, A)$

```
if $n$ = null then return ∅ ;
$R \leftarrow \emptyset$;
foreach $\langle r, m, s, e\rangle \in n$'s region list do
    if $b[r] = 1$ then
        foreach $o \in A[s, e]$ do
            if o is in $q.rng$ then
                $R \leftarrow R \cup \{o\}$;
return R
```

## C. Incrementally Search Keywords of An Input Prefix

**Range Query.** We divide query processing into two phases: (1) searching phase, in which we traverse the trie index using the query string then checked the spatial condition; and (2) result fetching phase, in which we access the data object array to locate and return results. Searching phase is run first. Given a query <*q.str*, *q.rng*>, we first converse *q.rng* according to the global space partitioning and obtain the regions intersected by *q.rng*. According to the regions obtained, an initialized bit array is obtained, by setting a bit to 1 if *q.rng* intersects a region; or 0, otherwise. We call the bit array *region status*. After that we begin to traverse the trie using *q.str*. When come across a trie node, we update the region status by a bitwise AND operation with the region bit array of the node. If a bit becomes 0 after the bitwise AND operation, it means that obviously there is no underlying object in this region for the query. Whenever query string cannot be matched or the intersected region status becomes all zero, we can terminate the traversal of the trie and return no results. We describe the above process in Algorithm 1. First, a region status is initialized (Line 1). Then it traverses the trie to match the incoming keystroke (Line 4) and update the region status (Line 5). If the keystroke cannot be matched or the region status becomes all zero, the whole traversal is terminated. It returns the currently located nodes and the region status for result fetching, or null to indicate there is no result (Line 14). The time complexity is $O(|q.str|)$, where || denotes the length of a string.

We show the result fetching phase in Algorithm 2. First, the region status bit array is scanned and the positions of the bits equal to 1 are obtained. Then we scan the corresponding regions in the region list. The objects in the data object array are located using the starting and ending pointers. Each object is verified by the query range. The time complexity is $\sum_{i=1}^{|L|}(e_i - s_i + 1)$, where $L$ denotes the region list, $s_i$ and $e_i$ denote the starting and ending pointers of the $i$-th entry in the list, respectively.

**Top-*k* Query.** As the algorithm framework of processing top-*k* queries is similar to processing range queries, except that the region status is not involved as there is no spatial constraint, we omit the detailed algorithms here. To efficiently process the top-k queries, several pruning techniques are proposed in our preliminary work [4] for the purpose of early termination.

## D. Supporting Fuzzy Search

Due to our trie-based index, our method can be easily extended to support any existing fuzzy search algorithm [5]-[10]. In the implementation of Loquat, we choose the trie-based method proposed in [6]. The basic idea of the method in [6] is to process the keystrokes in the query and compute a set of active nodes in the trie. We use edit distance as the threshold value to control the degree of fuzzy search, The path from the root to an active node is a string whose edit distance to the query is within the threshold $\tau$.

## E. Other Extensions

We discuss other extensions including supporting multiple-keyword search and synonym query autocompletion here. Compared to [3] and [1], our work is easier to extend to multiple keyword search. Because each traversal of one keyword in the trie will result in a temporary bit array. This temporary bit array can be used as a filter in the subsequent traversal of remaining keywords. If this preceding temporary bit array has no intersection with the subsequent bit array, the search can be terminated immediately because we can make sure that there are no intersected underlying objects located in any spatial grids. In addition, our index is also flexible to support synonym query autocompletion by simply adding synonym links between the trie nodes as shown in [11].

## F. Ranking Results

Although we give a general ranking function in Section 3, the popularity score can be aggregated by many ranking signals such as the user ratings, user feedback and price of the POI. In addition, we extend the ranking function by taking fuzzy threshold value into considerations. The renewed ranking function is modified as follow:

$$F(o,q) = \alpha \frac{o.scr}{max\_scr} + \beta\left(1 - \frac{\tau}{max\_\tau}\right) + (1-\alpha-\beta)\left(1 - \frac{dist(o.loc, q.loc)}{max\_dist}\right) \qquad (2)$$

In addition to $\alpha$, another weight $\beta$ is added to balance the three components. $\tau$ is the edit distance of matched query and data string pair. max_$\tau$ is the maximum threshold value allowed in the system. Using this new ranking function, an answer with smaller $\tau$ will be ranked higher as users always try to input correctly in the prefixes.

## IV. Experimental Results

## A. Setup

**Experimental Platform.** All experiments were done on a computer with an Intel i5 2.6GHz processor, 32GB RAM, running Ubuntu 14.04.1. The backend system modules are implemented using C++, the frontend interface is implemented using Node.js framework.

**Dataset.** Our experiments are conducted on two real datasets: UK and US. UK is a dataset containing POIs (e.g. banks and cinemas) in UK (www.pocketgpsworld.com). US is a dataset of 2M POIs located in US (www.geonames.org). The statistics are shown in Table I.

**Baseline method.** We choose the state-of-the-art PR-Tree as baseline [3]. PR-Tree is a tightly-combined method that merges trie and quadtree into a single index. It was designed for processing *k*nn queries.

**Evaluation measurements.** For each type of query, we generate 1,000 random queries by choosing strings that appear in the dataset. Longitude and latitude are normalized to [0, 1]. The default query range is a 0.08 × 0.08 square. The default value of *k* is 10. We measure (1) average query response time, including both searching time and result fetching time, (2) index construction time, and (3) index size.

TABLE I: Dataset Statistics

| Dataset | \|O\| | Size | Avg. str_len |
|---|---|---|---|
| UK | 181,549 | 7 MB | 10.1 |
| US | 2,234,061 | 82 MB | 10.6 |

## B. Range Queries

We first show the performance of processing range queries. Fig. 6 (a) – 6 (b) show the query processing times of the two algorithms on the two datasets, varying query string length. We can observe that query processing times decrease with the query string length, this is mainly because the number of results returned by both algorithms keeps reducing when query length grows which leads to fast processing time.

PR-Tree shows some rebounds when the query becomes longer because of more traversal cost. For both the small dataset UK and large dataset US, Loquat outperforms PR-Tree up to 10 times and 30 times, respectively.

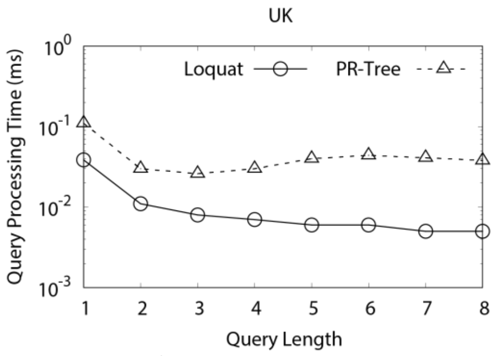

Fig. 6(a). Performance on range queries on UK.

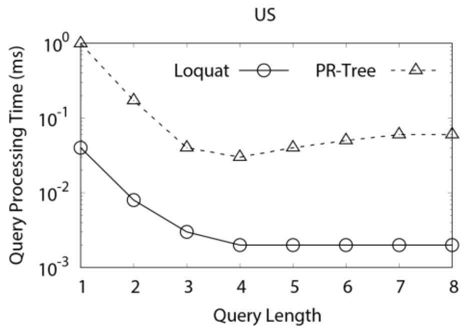

Fig. 6(b). Performance on range queries on US.

## C. Top-k Queries

For top-*k* queries, the comparison with two methods on the two datasets are shown in Fig. 7 (a) – 7 (b). Thank for the optimization techniques of Loquat, our system is always faster than PR-Tree, with the advantage of almost two orders of magnitude.

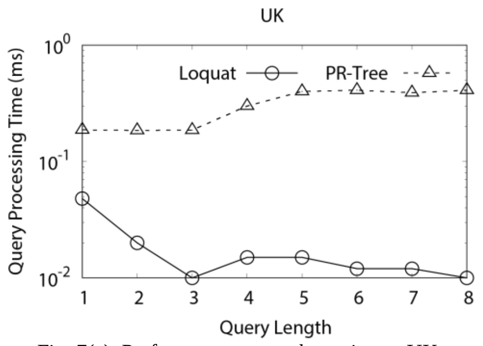

Fig. 7(a). Performance on top-*k* queries on UK.

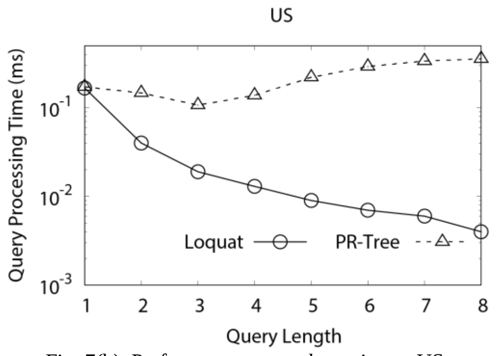

Fig. 7(b). Performance on top-*k* queries on US.

## D. Supporting Fuzzy Search

After extending to fuzzy search, the query processing time is shown in Figs. 8 (a) – 8 (b) for range queries and Figs. 9 (a) – 9 (b) for top-*k* queries. For both methods, because the search time of fuzzy search becomes a dominant component in the whole processing times, the times continue growing with the query string length. For range queries, Loquat achieves 15 times and 10 times faster on UK and US than PR-Tree. For top-*k* queries, Loquat also achieves 15 and 10 times faster than PR-Tree on UK and US, respectively.

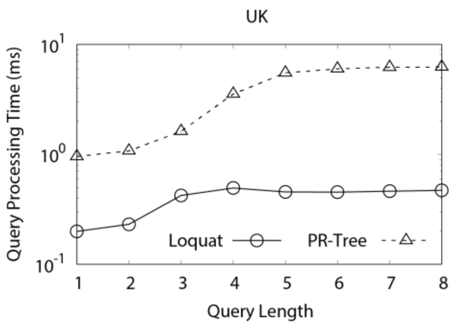

Fig. 8(a). Performance on fuzzy range queries on UK.

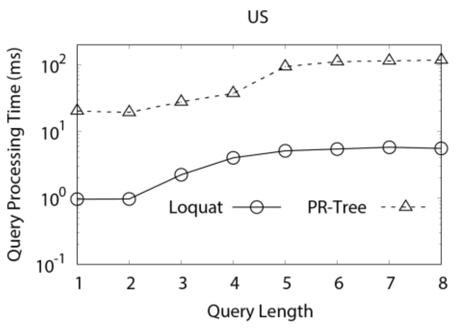

Fig. 8(b). Performance on fuzzy range queries on US.

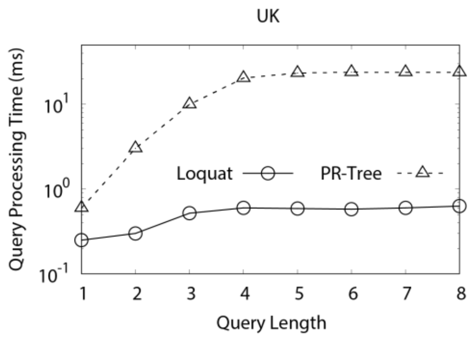

Fig. 9(a). Performance on fuzzy top-*k* queries on UK.

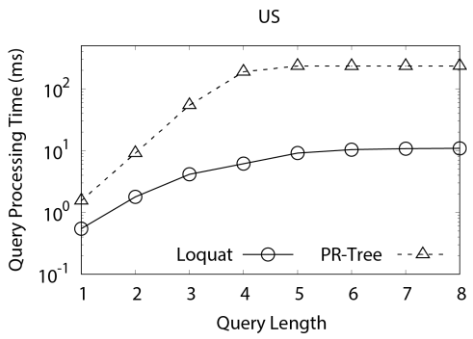

Fig. 9(b). Performance on fuzzy top-*k* queries on US.

## E. Index Construction

Table II shows the index sizes of the two methods on the two datasets. Table III shows the corresponding index construction times. Due to more information stored in the index, the size of Loquat is 2-3 times larger than PR-Tree. The construction time of Loquat is 2-3 times slower than PR-Tree. Both Loquat and PR-Tree can finish the index construction with an acceptable size and reasonable amount of time.

TABLE II: INDEX SIZE (MB)

| Dataset | Loquat | PR-Tree |
|---|---|---|
| UK | 74.0 | 39.8 |
| US | 1411.2 | 435.2 |

TABLE III: INDEX CONSTRUCTION TIME (SECONDS)

| Dataset | Loquat | PR-Tree |
|---|---|---|
| UK | 0.542 | 0.347 |
| US | 10.038 | 3.418 |

## V. RELATED WORKS

Query autocompletion has been widely adopted under various settings, including (1) location-aware type-ahead search, (2) fuzzy search or error-tolerant autocompletion and synonym autocompletion. (3) spatial-keyword search. Roy and Chakrabarti [1] studied the problem of location-aware type-ahead search and proposed a trie-based index which enumerates every query location possibility trying to rank the objects in advance. However, their index suffers from the consumption of large amount of memory. Ji *et al.* [2] proposed a method called Filtering-Effective Hybrid Indexing (FEH) to answer range queries and kNN queries. The method builds an R-tree to index data objects by their locations. Textual filters are used in each R-tree node to check whether the query string is a prefix of the objects in the subtree. After that, Zhong *et al*. proposed Prefix Region Tree (PR-Tree) [3] that considers textual and spatial partitioning simultaneously to build the index. Their main index is a special trie, whose each node is divided into four nodes, each representing a region in a quadtree, with centroids selected as the center for partitioning. The major problem of PR-Tree is the exhausive divisions of trie nodes cause too many branches of tree nodes, which makes the traversal very slow. Fuzzy type-ahead search or error-tolerant autocompletion were first studied in [5] and [6]. Li *et al.* [7] improved the method proposed in [5] for space and runtime performance. More efficient methods were proposed in [8]-[10]. After that, Xu *et al.* first studied the problem of synonym query autocompletion [11]. For spatial keyword search, this problem has been extensively studied in the database community, which is a problem about returning the relevant POIs considering both spatial proximity and textual relevance, when given a query composed of keywords and a location. Existing solutions are based on Rtree [12]-[16], grid [17], [18], and space filling curve [19]. We also refer users to an experimental evaluation [20] that compares these methods.

## VI. CONCLUSION

In this paper, we presented a new system for location-aware query autocompletion called Loquat, featuring with an interactive and user-friendly interface and several user-customized functions. Our system can answer range and top-*k* queries on a large scale. Our system can also be easily extended to support fuzzy search, multiple-keyword search and synonym autocompletion. The experiment results demonstrate the efficiency of Loquat and its superiority over existing state-of-the-art method. For future work, we plan to utilize external resources such as knowledge graphs or corpuses to support semantic search. We also plan to integrate our system with entity resolution techniques to support more accurate autocompletions.

## REFERENCES

[1] S. B. Roy and K. Chakrabarti, "Location-aware type ahead search on spatial databases: Semantics and efficiency," *ACM SIGMOD 2011*, pp. 361-372, 2011.

[2] S. Ji and C. Li, "Location-based instant search," in *Proc. of Int'l Conf. Scientific and Statistical Database Management (SSDBM 2011)*, 2011, pp. 17-36.

[3] R. Zhong, J. Fan, G. Li, K.-L. Tan, and L. Zhou, "Location-aware instant search," *ACM CIKM 2012*, pp. 385-394, 2012.

[4] S. Hu, C. Xiao, and Y. Ishikawa, "An efficient algorithm for location-aware query autocompletion," *IEICE Transactions*, vol. 101, no. 1, pp. 181-192, 2018.

[5] S. Ji, G. Li, C. Li, and J. Feng, "Efficient interactive fuzzy keyword search," *WWW 2009*, pp. 371-380, 2009.

[6] S. Chaudhuri and R. Kaushik, "Extending autocompletion to tolerate errors," *ACM SIGMOD 2009*, pp. 707-718, 2009.

[7] G. Li, S. Ji, C. Li, and J. Feng, "Efficient fuzzy full-text type-ahead search," *VLDB J.*, vol. 20, no. 4, pp. 617-640, 2011.

[8] C. Xiao, J. Qin, W. Wang, Y. Ishikawa, K. Tsuda, and K. Sadakane, "Efficient error-tolerant query autocompletion," *PVLDB*, vol. 6, no. 6, pp. 373-384, 2013.

[9] X. Zhou, J. Qin, C. Xiao, W. Wang, X. Lin, and Y. Ishikawa, "BEVA: An efficient query processing algorithm for error-tolerant autocompletion," *ACM Trans. Database Syst.*, vol. 41, no. 1, pp. 1-5, 2016.

[10] D. Deng, G. Li, H. Wen, H. V. Jagadish, and J. Feng, "META: An efficient matching-based method for error-tolerant autocompletion," *PVLDB*, vol. 9, no. 10, pp. 828-839, 2016.

[11] P. Xu and J. Lu, "Top-k string auto-completion with synonyms," *DASFAA*, pp. 202-218, 2017.

[12] I. D. Felipe, V. Hristidis, and N. Rishe, "Keyword search on spatial databases," *ICDE 2008*, pp. 656-665, 2008.

[13] A. Cary, O. Wolfson, and N. Rishe, "Efficient and scalable method for processing top-k spatial boolean queries," in *Proc. of Int'l. Conf. Scientific and Statistical Database Management (SSDBM 2010)*, 2010, pp. 87-95.

[14] Z. Li, K. C. K. Lee, B. Zheng, W. C. Lee, D. Lee, and X. Wang, "Ir-tree: An efficient index for geographic document search," *IEEE TKDE*, vol. 23, no. 4, pp. 585-599, 2011.

[15] D. Wu, G. Cong, and C. S. Jensen, "A framework for efficient spatial web object retrieval," *VLDB J.*, vol. 21, no. 6, pp. 797-822, 2012.

[16] D. Wu, M. L. Yiu, G. Cong, and C. S. Jensen, "Joint top-k spatial keyword query processing," *IEEE TKDE*, vol. 24, no. 10, pp. 1889-1903, 2012.

[17] S. Vaid, C. B. Jones, H. Joho, and M. Sanderson, "Spatio-textual indexing for geographical search on the web," *Int'l. Symp. Spatial and Temporal Databases (SSTD 2005)*, pp. 218-235, 2005.

[18] A. Khodaei, C. Shahabi, and C. Li, "Hybrid indexing and seamless ranking of spatial and textual features of web documents," *DEXA 2010*, vol. 6261, pp. 450-466, 2010.

[19] M. Christoforaki, J. He, C. Dimopoulos, A. Markowetz, and T. Suel, "Text vs. space: Efficient geo-search query processing," *ACM CIKM 2011*, pp. 423-432, 2011.

[20] L. Chen, G. Cong, C. S. Jensen, and D. Wu, "Spatial keyword query processing: An experimental evaluation," *PVLDB*, vol. 6, no. 3, pp. 217-228, 2013.

**Sheng Hu** is a Ph.D candidate in Graduate School of Information Science, Nagoya University. He received B.E. degree from North China Electric Power University in 2013. His research interests include textual databases and spatiotemporal databases.

**Chuan Xiao** is an assistant professor in Graduate School of Information Science, Nagoya University. He received B.E. degree from Northeastern University, China in 2005, and Ph.D. degree from The University of New South Wales in 2010. His research interests include data cleaning, data integration, textual databases, and graph databases. He is a member of DBSJ.

**Ishikawa Yoshiharu** is a professor in Graduate School of Informatics, Nagoya University. He received B.S., M.E., and Dr. Eng. degrees from University of Tsukuba in 1989, 1991, and 1995, respectively. His research interests include spatio-temporal databases, mobile databases, sensor databases, data mining, information retrieval, and e-science. He is a member of ACM, DBSJ, IEEE, IEICE, IPSJ, and JSAI.