LINQ: A Framework for Location-aware Indexing and Query Processing

Xiping Liu, Lei Chen, *Member, IEEE,* and Changxuan Wan

Abstract—This paper studies the *generic location-aware rank query* (GLRQ) over a set of location-aware objects. A GLRQ is composed of a spatial location, a set of keywords, a query predicate, and a ranking function formulated on location, text and other attributes. The result consists of k objects satisfying the predicate ranked according to the ranking function. An example is a query searching for the restaurants that 1) are nearby, 2) offer "American" food, and 3) have high ratings (rating $>$ 4.0). Such queries can not be processed efficiently using existing techniques. In this work, we propose a novel framework called LINQ for efficient processing of GLRQs. To handle the predicate and the attribute-based scoring, we devise a new index structure called *synopses tree*, which contains the synopses of different subsets of the dataset. The synopses tree enables pruning of search space according to the satisfiability of the predicate. To process the query constraints over the location and keywords, the framework integrates the synopses tree with the spatio-textual index such as IR-tree. The framework therefore is capable of processing the GLRQs efficiently and holistically. We conduct extensive experiments to demonstrate that our solution provides excellent query performance.

Index Terms—Location-aware, rank query, synopses tree, LINQ

1 Introduction

With the proliferation and widespread adoption of mobile telephony, it is more and more convenient for users to capture and publish geo-locations. As a consequence, more and more location-aware datasets have been created and made available on the Web. For example, Flickr, one of the biggest photo-sharing website, has millions of geo-tagged items every month [1]. The popularity and large scale of the location-aware datasets make location-aware queries important.

In this work, we are interested in location-aware rank query, an important class of location-aware query. Examples of location-aware rank query include the (k-) nearest neighbor (NN) query and location-aware keyword query (LKQ, a.k.a spatial keyword query) [1], [2], [3], [4], [5] and [6]. NN queries and LKQs have wide applications in many domains. However, there are a lot of location-aware datasets that demand more powerful and flexible location-aware rank queries. Fig. 1 shows an example of a restaurant on Yelp [2]. For the restaurant, Yelp gives its location ("1429 Mendell St, San Francisco, CA 94124"), categories ("Soul Food, American (Traditional), Music Venues"), rating (4.5 stars) as well as the number of reviews (150). Another example is the photos on Flickr, where in addition to the geo-location and text, each photo also has some numeric attributes, such as the number of views, the number of favorites, the number of comments, and so on. The restaurants on Yelp and photos on Flickr are mixtures of location, text and other types of information. They are much more complex than spatio-textual objects. We term these objects as *location-aware objects*. For location-aware objects, simple NN queries and LKQs may not be expressive enough to find the objects of interests. For example, on the restaurant dataset, through LKQs, one can find the most nearest and relevant restaurants. But one may hope to find the nearest and relevant restaurants satisfying certain conditions, such as no-smoking, healthy, or top-ranked. On the Flickr dataset, users may want to fetch the relevant and nearest photos that are highly-rated. On these datasets, people may wish to search by not only location and keywords, but also conditions on other attributes.

In this paper, we study the *generic location-aware*

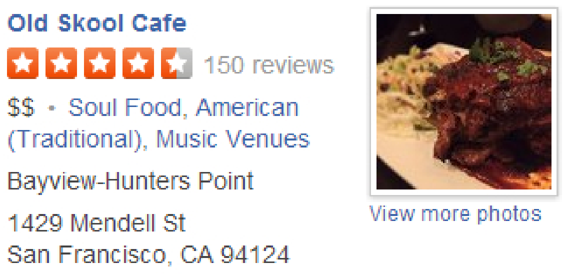

Old Skool Cafe
150 reviews
$$ • Soul Food, American (Traditional), Music Venues
Bayview-Hunters Point
1429 Mendell St
San Francisco, CA 94124
View more photos

Fig. 1. A snippet about a restaurant on Yelp.com

• *Xiping Liu and Changxuan Wan are with the Jiangxi University of Finance and Economics, Nanchang 330013, China. E-mail: lewislxp@gmail.com, wanchangxuan@263.net*

• *Lei Chen is with the Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong, China. E-mail: leichen@cse.ust.hk.*

1. http://www.flickr.com/map/
2. http://www.yelp.com

rank query (GLRQ) over a set of location-aware objects. A GLRQ is composed of a query location, a set of keywords, a query predicate, and a ranking function combining spatial proximity, textual relevance and other measures (e.g. certain attribute values). The result of the query consists of k objects that satisfy the predicate, ranked according to the specified ranking function. Obviously, the GLRQ has the NN query and LKQ as special cases.

Example 1: An example (Q_1) of the GLRQ is to search restaurants near a certain location (denoted by loc) that provide "American traditional" food with the constraint that the ratings should be above 4.0, and rank the restaurants based on spatial proximities and textual relevances. The query intent cannot be expressed by a LKQ, because the predicate $\mathsf{rating} > 4.0$ cannot be processed by simple keyword matching. If we change the ranking function to consider also the ratings of objects, we get another query Q_2, which is also a GLRQ. Another example query (Q_3) requests hotels that are near a location and have popularities above 8.5. We can see that a GLRQ allows users to specify preferences on spatial, text and other attributes, thus is more powerful and flexible than NN query and LKQ. We believe that this query will be very helpful for users in many applications.

Up to now, we have not found any work devoted to the generic form of location-aware rank query. One may think that the GLRQs can be simply processed based on the existing techniques. However, that is not the case. Consider again the query Q_1 shown in Example 1. A straightforward approach to process the query is as follows: incrementally compute the results of a LKQ $Q'(loc,$ "American traditional"$)$ using existing techniques, then validate each object obtained against the predicate ($\mathsf{rating} > 4.0$), and finally return the objects satisfying the predicate in a ranked order. In this approach (called *LKQ-first* approach), to get top-k results, it is unavoidable to compute much more intermediate results than k. For example, suppose only 1% restaurants have ratings above 4.0, and the query requests 3 results. Then the probability of obtaining 3 final results after generating even 100 LKQ results is still 0.0794 (more analyses will be presented later). Another possible approach is to put everything in a relational database, and process the query within RDBMS. In this approach, RDBMS will first obtain all the candidates satisfying the predicate, and then score the candidates. We assume the selectivity of the query predicate is ρ, and a secondary index (B$^+$-tree) has been built on the attribute rating. Then, the cost (in terms of number of disk access) of this approach is about $HT + LB * \rho + n_r * \rho$ [7], where HT is the height of the B$^+$-tree, LB is the number of leaf blocks in the index, and n_r is the total number of objects. As n_r can be a very large number, the cost is very high. Therefore, RDBMS cannot provide satisfactory query performance. For the query Q_2 in Example 1, LKQ-first does even worse, because a top LKQ result, even satisfying the query predicate, may not be a top result of Q_2. As for query Q_3, existing LKQ techniques do not help at all, because it does not contain keywords. Although RDBMS can process Q_2 and Q_3, as analyzed before, the efficiency is very low.

In this paper, we propose a novel framework, called LINQ (Location-aware INdex and Querying framework), to efficiently process the GLRQ. A core component of the framework is a new index structure called *synopses tree*, which stores synopses of different sets of objects in a tree. The synopses tree can be combined with spatial index, e.g. R-tree, text index, e.g. inverted file, or spatio-textual index, e.g. IR-tree [2], to provide integrated indexing and querying capability. Based on the LINQ framework, we present the search algorithm, especially estimation algorithm, for GLRQ.

In summary, we make the following contributions in this paper:

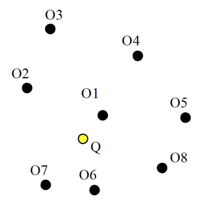
- We formulate the generic location-aware rank query (GLRQ), which retrieves the objects satisfying a query predicate, ranks and returns the results based on spatial proximity, textual relevances and measures obtained from attribute values.
- We develop a novel framework called LINQ, for efficient indexing and querying of GLRQs. LINQ develops the synopses tree to utilize synopses of non-spatial attributes, and combines the synopses tree with other indexes to index and query the GLRQ.
- We conduct extensive experiments on both real and synthetic datasets, and the results demonstrate the effectiveness and efficiency of our method.

The rest of the paper is organized as follows. Section 2 formally defines the problem. Section 3 introduces several possible solutions. Section 4 presents the LINQ framework. Empirical performances of our proposal are evaluated in Section 5, and related work is discussed in Section 6. Finally, we conclude the work in Section 7.

2 Problem Statement

We denote a location-aware object O as a triple $(\lambda, W, \mathcal{A})$, where $O.\lambda$ is a location descriptor, $O.W$ is a set of keywords, and $O.\mathcal{A} = (O.A_1, O.A_2, \cdots)$ is a set of attributes. We use $O.A_i$ to denote the value of O on attribute A_i. Without loss of generality, we assume the attributes in $O.\mathcal{A}$ are numeric attributes.

Example 2: We use a dataset about restaurants as an example throughout this paper. In this dataset, $\mathcal{A}$ = (rating, $\mathsf{\#reviews}$, health), denoting the rating, the number of reviews, and the health score of the restaurant, respectively. All these attributes are borrowed from Yelp. Fig. 2 shows several instances of object. The locations of the objects are implied by their positions



ID	W	rating	#reviews	health
O1	American Traditional	4.5	1500	96
O2	Korean, Chinese	5	160	90
O3	American New	4.5	3306	95
O4	Italian	4.5	610	96
O5	French, American New	4.0	1671	100
O6	American Traditional	3.5	1650	83
O7	American New	4.0	204	90
O8	Chinese	2.5	121	85

Q: American, rating >= 4.0, health >= 95

Fig. 2. A set of location-aware objects

on the plane, and the attributes and keywords are shown in the table.

In this work, we are interested in a type of query, called *generic location-aware rank query* (GLRQ), which allows users to specify constraints on spatial, text and other attributes, and returns top-k results according to user-provided ranking function.

Formally, a GLRQ Q is a quintuple (λ, W, P, f, k), where $Q.\lambda$ is the location descriptor of Q, and $Q.W$ is a set of keywords. $Q.P$ is a query predicate of the form $P_1 \wedge P_2 \wedge \cdots \wedge P_n$, where P_i is a simple selection predicate specified on a single attribute. The fourth component $Q.f$ is a ranking function. We assume each attribute, including location and text, has a scoring function, measuring the "goodness" of an object in terms of the attribute. The scoring function on location returns spatial proximity, whereas the function on text measures the textual relevance. For other attributes, we assume a monotone or anti-monotone function is defined. Then the ranking function $Q.f$ combines the scores of individual attributes. Specifically, $Q.f$ is defined as follows.

$$f(O,Q) = \begin{cases} 0 & \neg Q(O) \\ g(prox(O,Q),\ rel(O,Q), & \\ \qquad s_1(O.A_1),\ s_2(O.A_2), \cdots) & \text{otherwise} \end{cases} \tag{1}$$

where $\neg Q(O)$ means that O does not satisfy $Q.P$. $prox(O,Q)$ is the spatial proximity of $O.\lambda$ and $Q.\lambda$. In this work, the proximity is computed in an Euclidean space. $rel(O,Q)$ is the textual similarity between $O.W$ and $Q.W$. $s_1(\cdot), s_2(\cdot), \cdots$ are the scoring functions on the attributes $O.A_1, O.A_2, \cdots$. We follow the conventions in existing work [2], [4], [5] to assume that g is a monotone function. The last component, $Q.k$, is the number of results requested.

Putting all things together, a GLRQ Q returns $Q.k$ objects that satisfy query predicate $Q.P$ and have greatest scores according to ranking function $Q.f$. In formal, the result of a GLRQ Q on dataset D, $Q(D)$, is a subset of D with $Q.k$ objects satisfying

$$\forall O' \in (D - Q(D))(\forall O \in Q(D)(f(O',Q) \leq f(O,Q))$$

Example 3: Consider the dataset in Fig. 2. An example query is Q_4 where W={American}, P=(rating $\geq$ 4.0 $\wedge$ health $\geq$ 95), $k = 3$, and the function f (the "otherwise" case) is defined as

$$f(O,Q) = w_1 \cdot prox(O,Q) + w_2 \cdot rel(O,Q) + w_3 \cdot \frac{O.rating}{5.0} + w_4 \cdot \frac{O.health}{100}$$

where $w_1 + w_2 + w_3 + w_4 = 1$. The query object is depicted in yellow and other objects are depicted in black in Fig. 2. The results of the query should be O_1, O_5 and O_3.

Some important notations used in the paper are summarized in Table 2.

TABLE 1
Notations

Symbol	Meaning
$O = (\lambda, W, \mathcal{A})$	an object O described by a location λ, a set W of keywords , and a set $\mathcal{A}$ of attributes
$Q = (\lambda, W, P, f, k)$	a query Q described by a location λ, a set W of keywords, a query predicate P, a ranking function f, and the number k of results
N (or e)	a node (or an entry) in the synopses tree
$N.syn$ (or $e.syn$)	the synopses corresponding to N or e
$N.D$ (or $e.D$)	the dataset corresponding to N (or e)
$Q(N)$	the result of evaluating $Q.P$ on N
$\mathcal{H} = \{H_1, H_2, \cdots\}$	$\mathcal{H}$ is a set consisting of global histograms $H_1, H_2, \cdots$
B_{ij}	a bucket in the histogram H_i
$\mathcal{B}$	the set of global buckets, with cardinality $\lvert\mathcal{B}\rvert$, consisting of all buckets from the global histograms

3 Possible Solutions

In this section, we discuss several possible solutions towards the problem.

Naive LKQ method (LKQ-naive). This method uses the techniques developed for location-aware keyword query (LKQ). In other words, it treats all attribute values as keywords. Take the query in Example 3 as an example. In this method, the query is just transformed to a standard LKQ Q', where $Q'.W$ = {American, traditional, 4.0, 4.1, $\cdots$, 96, 97, $\cdots$}. The method has several shortcomings. First, the index size is significantly increased, because the vocabulary of keywords is greatly expanded. Second, the query is complicated, and the method has to scan a large number of indexes (e.g. inverted files), incurring a great computation cost. Last but not the least, keywords are typically indexed by inverted files (such as the work in [5], [2]), but traditional queries (e.g. SQL) on these attributes usually require B$^+$-trees. As a result, the method cannot provide adequate support for these traditional queries, unless B$^+$-trees are also built on these attributes, but that will lead to great redundancy.

Another variant is to treat ranges of values as keywords. This variant will decrease the index size, as the number of keywords is smaller, but it still has the above-mentioned problems. In addition, it has increased complexity of splitting the values. It is very difficult to decide the best way to create the ranges. What is more, the query processing is not straightforward. Just consider the predicate health $\geq$ 75. As the predicate cannot be perfectly transformed into a set of keywords, some post processing stages are necessary, such as result verification.

LKQ-first method (LKQ-F). This method first computes the results of the query without predicate (an LKQ query), and then validates the results obtained against the query predicate.

If the ranking function consists of only spatial proximity and textual relevance, the efficiency of LKQ-F depends on the selectivity of the query predicate. Let $sel(Q.P)$ be the selectivity of the query predicate $Q.P$, where the selectivity is defined as the proportion of objects matching the predicates in all the objects. Then given an object O, the possibility of O satisfying $Q.P$ is $sel(Q.P)$. Let x be the number of additional results needed to obtain the final top-k results. The value of x follows the *negative binomial distribution* with the probability density function as follows

$$P_{k,p}(x) = \binom{x+k-1}{k-1} \cdot p^k \cdot (1-p)^x$$

where $p = sel(Q.P)$. For example, suppose $p = 0.01$ and $k = 3$, then the probability of obtaining 3 results satisfying the query predicate after generating even 100 results is still 0.0794.

If attributes other than the location and text are involved in the ranking function, the efficiency of LKQ-F is even worse. That is because LKQ-F does not take the attribute scores into account when generating the intermediate LKQ results. Consequently, more intermediate results are generated, until it is guaranteed that no better results can be found.

If the query does not contain keyword constraints, LKQ-F does not apply at all.

Therefore, the LKQ-F method is not a suitable solution.

Predicate-first method (PF). In this approach, the query predicate is evaluated to get a set of candidates, which are then scored according to the ranking function, and finally the k objects with greatest scores are returned. As discussed in Section 1, this method generally has a very high cost.

R-tree based method (RT). Another approach is to index the location and other attributes in one index, such as R-tree. For the query Q_4 in Example 3, if rating and health are also indexed together with the locations, the query can be processed easily. This approach has several shortcomings. First, as the R-tree has to index all the attributes, objects are nearly duplicated in the R-tree, and the redundancy would be very high. Second, the R-tree must have a high dimensionality. Due to the curse of dimensionality [8], the method can hardly provide satisfactory performance.

It should be noted that although the location-aware objects can be well stored in a relational database, the current support for LKQ, not to mention the GLRQs, in RDBMS is very poor. In fact, we have loaded our experimental data into MySQL and a commercial RDBMS, built all the necessary indexes (spatial indexes on the locations, full-text indexes on the texts, and B$^+$-tree indexes on other attributes), and then posted several LKQs and GLRQs in SQL language on the databases. The observed query plans show that both RDBMSs process the LKQs in a rather naive way (direct sorting), and the GLRQs in the predicate-first way. Therefore, the built-in capabilities of RDBMS are far from satisfactory.

According to the semantics of GLRQs, objects are not returned as final results if they do not satisfy the query predicate, or they have low scores. Therefore, we can prune the results using two strategies: predicate-based pruning and score-based pruning, which prune the results based on the satisfiability and scores, respectively. The LKQ-F and PF solutions are not efficient because they conduct the predicate-based pruning and score-based pruning in separate stages, leading to large numbers of intermediate results. To achieve good query performance, it is essential to prune the objects that are not in the top-k results as early as possible. In this work, we propose a framework to index and process the GLRQ, which combines predicate-based pruning and score-based pruning in a systematic manner.

Without loss of generality, we assume that the spatial locations have been indexed by an R-tree, and other attributes have been indexed by B$^+$-trees.

4 The LINQ Framework

In this section, we present the LINQ framework to support the indexing of location-aware objects and query processing of GLRQs. In the framework, there is a synopses tree summarizing numeric attributes, and a spatio-textual index built on locations and keywords. We first introduce the index structure, and then the query processing.

4.1 Synopses Tree

It is very common for a DBMS to build synopses on top of one or multiple attributes. The synopses are primarily used to estimate the selectivities of queries so that better query plans can be found. In this work, we also make use of synopses, and the motivation is similar: we use the synopses to estimate the satisfiability of the query predicate. As the results cannot be obtained in one shot, we build a synopses tree, instead of a synopsis, to guide the search.

Basically, the synopses tree is a tree of synopses, where each node represents a synopsis of a dataset. In the synopses tree, each leaf has a number of pointers to objects. Each internal node has a number of entries, where each entry consists of a synopsis and a pointer to a child node. Each node or entry encloses a set of objects. We can imagine that there is a dataset associated with each node or entry, which contains all objects enclosed. Let N (e) be a node (an entry) in the synopses tree. We use $N.D$ ($e.D$) to denote the imaginary dataset corresponding to N (e), and $N.syn$ ($e.syn$) the synopsis of $N.D$ ($e.D$). In a synopses tree, if a node N_1 is a child of N, then $N_1.D \subseteq N.D$; if $N_1, \cdots, N_n$ are the children of N, then $N.D = N_1.D \cup \cdots \cup N_n.D$. Given a dataset D, let ST be the synopses tree built over D, then the root of ST summarizes the whole dataset D, and each node summarizes a subset of D. What is more, the nodes near the root provide summaries of larger subsets of the dataset.

The strength of synopses tree is that, as each entry in a non-leaf node contains a synopsis, we can estimate the satisfiability of a query predicate when visiting the entry, making it possible to do predicate-based pruning. We will show in detail how to process GLRQs later on.

4.2 Design of Synopses Tree

In the following, we give more details about the synopses tree.

4.2.1 *Factorized multi-dimensional histogram*

There are different families of synopses, such as random samples, histograms, wavelets and sketches [9]. We use histograms in this work, as histograms have been extensively studied and have been incorporated into virtually all RDBMSs.

As more than one attributes are concerned, we build multi-dimensional histograms to summarize the datasets. Multi-dimensional (nD) histogram has been widely used to summarize multi-dimensional data. Basically, an nD histogram is obtained by partitioning the multi-dimensional domain into a set of hyper-rectangular buckets, and then storing summary information for each bucket. As a synopsis, nD histogram is able to provide accurate approximation, but it is very expensive to construct and maintain an nD histogram. To reduce the complexity of nD histogram, we use the graphical model of Tzoumas et al [10] to factor the nD data distribution into two-dimensional (2D) distributions. The effectiveness of this method has been verified. By this way, we can provide accurate approximations of the joint data distributions with low cost.

The graphical modeling of a dataset generates a model called *junction tree*. In our setting, the junction tree is a tree structure where each node is a pair of attributes. Then, according to the junction tree, the joint distribution of multiple attributes can be factorized into a set of 2D distributions. An example of junction tree is shown below. A more detailed explanation of the modeling method is presented in Appendix A.

Example 4: Take the data in Fig. 2 as an example. The method first builds a model called moral graph (e.g. Fig. 3(a)), where each node represents an attribute, and each edge represents a dependency between a pair of attributes. Then, a junction tree is created based on the moral graph, where an attribute is put together with the attribute it depends on. The junction tree is shown in Fig. 3(b). According to the junction tree, we keep distributions of {#reviews, health} and {#reviews, rating}. Other distributions can be derived from the two distributions. In other words, we keep two 2D histograms instead of a 3D histogram.

4.2.2 *Global buckets*

The work of [10] addresses the problem of how to construct a single nD histogram over a dataset. In our work, we need to build a tree of synopses. If the nD-histograms in the synopses tree are constructed independently, the cost would be too high. In the following sections, we propose a method to construct the synopses with reduced cost.

We notice that, the synopses in the synopses tree are not independent. First, the datasets of different nodes may be similar or overlapping. Second, the dataset of a node at a higher level is a superset of that of its descendants. Considering the characteristics above, we construct the histograms in a holistic way. Specifically, we construct a set $\mathcal{H} = \{H_1, H_2, \cdots\}$ of global histograms for the whole dataset D, where $H_i = \{B_{i1}, B_{i2}, \cdots\}$ is a global histogram, and B_{ij} is a bucket the histogram H_i. The set of global histograms is used by all entries in the synopses tree. Let $\mathcal{B}$ be the set of buckets in all histograms, i.e. $\mathcal{B} = \bigcup_i H_i$, and $|\mathcal{B}|$ be the number of buckets in $\mathcal{B}$. For any entry e, an array $e.b$ with $|\mathcal{B}|$ elements is maintained, where each element in the array is the statistics about the $e.D$ in a bucket. Thus the array records the summary statistics of $e.D$.

Example 5: Consider the Example 4. We construct histograms for the 2D datasets (#reviews, health) and (#reviews, rating). The global histograms are shown in Fig. 4(a) and (b). Here we assume 8 buckets are available, and the equi-depth bucketing scheme is adopted. The right part of Fig. 4 shows the local information kept by each entry, where $R_1, \cdots, R_6$ are the entries in the R-tree (the R-tree is presented in Fig. 5). In this example, we just keep the count of objects falling in each bucket (more discussions will be presented later).

By using global histogram definitions, i.e. global buckets, the construction cost and storage overhead of histograms are decreased. This is because the global histograms are constructed and stored only once. For each entry in a non-leaf node, only some local statistics are kept.

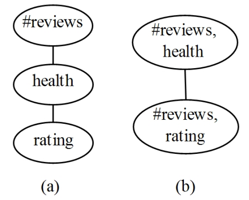
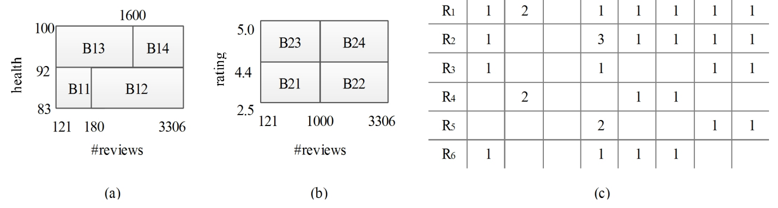


Fig. 3. An example of junction tree. (a) The moral graph, (b) The junction tree



	B_{11}	B_{12}	B_{13}	B_{14}	B_{21}	B_{22}	B_{23}	B_{24}
R_1	1	2		1	1	1	1	1
R_2	1			3	1	1	1	1
R_3	1			1			1	1
R_4		2			1	1		
R_5				2			1	1
R_6	1			1	1	1		

Fig. 4. The histograms constructed for the example dataset. (a) H1, (b) H2, (c) local information

4.2.3 Compact local information

A bucket in the global histograms stands for a hyper-rectangle within the domain of data. To approximate the data points falling in the hyper-rectangle, some statistics about the distribution of data in the rectangle are needed. In Example 5, for each entry, we keep the count information regarding each bucket. In fact, as we are only concerned with the satisfiability of query predicate in this work, some simple statistics is enough.

For example, consider the 2D bucket B_{11} in Fig. 4, it corresponds to a rectangle $\{121 \leq \#\text{reviews} \leq 179, 83 \leq \text{health} \leq 91\}$. Suppose we are interested in objects satisfying $P_1 : \text{health} \geq 85$. If we know that the bucket B_{11} is not empty, then we can estimate that the predicate P_1 holds on the bucket. From this example, we can see that it is possible to do estimation with only one bit kept for a bucket, indicating whether the bucket is empty or not. However, the estimation error would be high if the bucket covers wide ranges. To improve the accuracy of estimation, more in-bucket information is needed.

In this work, we use a compact bit-based representation of the local information in each bucket. We split each bucket into M partitions, and then stores a M-bit string where each bit is 1 if the corresponding partition is not empty or 0 otherwise. According to the practice of [11], 32 and 64 are good choices for M on 32-bit and 64-bit platforms, respectively. By this method, the accuracy of synopsis is improved by an order of M. In our implementation, we choose $M = 32$.

The M partitions are obtained by successive binary splits of dimensions in a round-robin manner. For example, if the buckets have 3 dimensions x_1, x_2, x_3, and $M = 32$, then one split order is x_1, x_2, x_3, x_1, x_2. Each split is done on the middle, so that the partitions are equally sized. To interpret the 32-bit array when estimate, the boundaries of the partitions should be known. For this purpose, we keep the splitting information, and the correspondence between the bits and the partitions. As all the buckets are split in the same way, this information can be stored only once as background information.

Example 6: Still consider the 2D bucket $B_{11} : \{121 \leq \#\text{reviews} \leq 179, 83 \leq \text{health} \leq 91\}$ in Fig. 4. According to the splitting method, the #reviews dimension will be split three times into 8 ranges. and dimension health will be split twice. We can define a default order on the partitions, e.g., (#reviews=[121, 127], health=[83, 84]), (#reviews=[121, 127], health=[85, 86]), $\cdots$. If the first two bits of a bit string is "10", it indicates that there are points in the first partition while not in the second partition.

4.2.4 Summary of the synopses tree

The key points in the design of synopses tree is summarized as follows.

- Conceptually, each synopsis is composed of two parts: the synopsis definition, and the statistics about data. Actually, the synopsis definition is the same for all entries, so there is only one copy of global synopsis definition.
- Each synopsis is an nD histogram, which is decomposed into multiple 2D histograms. In the global, the boundaries of the buckets, and the split information are maintained.
- The local description of an entry is a set of simple M-bit strings, each corresponding to a bucket in a 2D histogram. The data distribution of each bucket is interpreted on the basis of global histogram when necessary.

Example 7: The synopses tree of the dataset in Fig. 2 is illustrated in Fig. 5. As discussed earlier, two 2D histograms H_1 and H_2 are constructed, each with 4 buckets. For each entry in a non-leaf node, a synopsis is maintained, which keeps information about the buckets in the global histogram. For example, to describe the data distribution of an entry R_6 with respect to B_{11}, a 32-bit string is used ($M = 32$). Therefore, the synopsis of an entry is an array of bit strings corresponding to the buckets in the global histograms.

We estimate the space consumption of the synopses. Maintaining the boundaries of a bucket needs at most 4 numeric values, i.e. 32 bytes. Thus the global histogram requires $32 \cdot |\mathcal{B}|$ bytes. For each entry in the non-leaf node, $32 \cdot |\mathcal{B}|$ bits or $4 \cdot |\mathcal{B}|$ bytes are needed (assume $M = 32$). The total space of the synopses is $32 \cdot |\mathcal{B}| + 4 \cdot |\mathcal{N}| \cdot |\mathcal{B}|$ bytes, where $|\mathcal{N}|$ is the number of

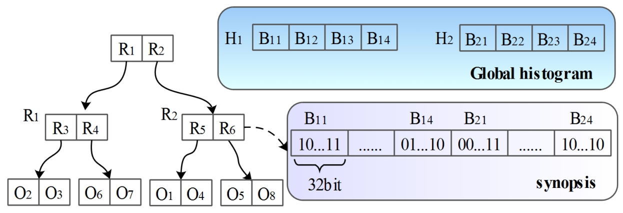


Fig. 5. An example of synopses tree

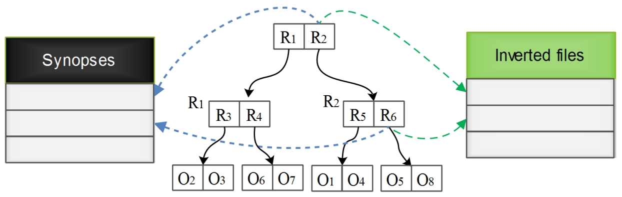


Fig. 6. The structure of the combined index

non-leaf entries in the tree. Obviously, the overall cost is dominated by the second part. To reduce the space cost, we can reduce the number of buckets ($|\mathcal{B}|$) in the global histograms. Note that each bucket is divided to 32 partitions, so with $|\mathcal{B}|$ buckets, we obtain $32 \cdot |\mathcal{B}|$ partitions. The accuracy of the synopsis is proportional to the number of partitions, therefore, we can choose a moderate number of buckets in the global histograms. For example, from 32 buckets we can get 1024 partitions, a practical number for estimation. Another optimization is that the synopsis of an entry is represented in bit strings, which can be compressed to save space. In fact, the synopsis of an entry far from the root will have very sparse bit string (many "0"s), thus can achieve a high compression rate. In our implementation, the bit strings are compressed using EWAH technique [12], which demonstrates a high compression rate. Note that the compression will not introduce any noises. Using these methods, we are able to reduce the space consumption to a low level.

4.3 Combining Synopses Tree with Other Indexes

The synopses tree summarizes the distribution of numeric attributes. It is able to address part of the GLRQ processing problem. For example, for the query in Example 3, the synopses tree can be used to find the objects that satisfy rating $\geq 4.0 \wedge$ health ≥ 95, and have greatest partial scores according to the function $0.05 \times \frac{O.rating}{5.0} + 0.05 \times \frac{O.health}{100}$. To answer the GLRQ, we need to combine synopses tree with other indexes on locations and texts.

The state-of-the-art index structures supporting locations and texts are the IR-tree family [2], [3] and S2I family [5]. The basic structure of IR-tree index is an R-tree, where each entry is associated with an inverted file. Our synopses tree can be easily combined with IR-tree index. Fig. 6 illustrates the combined index structure, for the dataset shown in Fig. 2. In the middle of the figure there is an R-tree indexing the locations of the objects. Each entry in the R-tree is associated with two additional components: the inverted file from the IR-tree, and the synopsis from the synopses tree. The inverted files and the synopses are stored separately from the R-tree. This structure is flexible in that the structure of the R-tree is not influenced by other parts, and the R-tree can be queried alone, with or without the inverted files and the synopses.

S2I maps each distinct term to an aggregated R-tree (aR-tree) [13] or block file, depending on whether the term is a frequent term or not. It is possible to merge the aR-tree with synopses tree. However, that will significantly increase the size of the index, making it unaffordable. It is very difficult, if not impossible, to combine the synopses tree with the block file, because the synopses tree is a hierarchical data structure, while the block file is a plain flat file. In the following, we integrate the synopses tree with IR-tree to process the GLRQ.

4.4 Query Processing

In this section, we discuss the query processing algorithms for the GLKQs.

4.4.1 *Algorithm framework*

The algorithm framework (Algorithm 1) exploits the well-known best-first strategy [14] to search the combined index. In the algorithm, a priority queue $Queue$ is used to keep track of the nodes and objects to explore in decreasing order of their scores. $maxf(e, Q)$ is the maximum matching score of entry e to query Q, which are used as the keys of entries in the queue. The computation of $maxf(\cdot, \cdot)$ will be explained later. $TopK$ keeps the current top-k results. The R-tree is traversed in a top-down manner. At each node N, if N is an internal node instead of a leaf (line 10), for each entry e in node N, the algorithm estimates $maxf(e, Q)$. If $maxf(e, Q) > 0$, it implies that there may be objects enclosed by e satisfying the query predicate, so e with $maxf(e, Q)$ is added to the queue. If N is a leaf, it computes the score of each entry (object) (line 19), and pushes the entries with non-zero scores to the queue. If N is an object, N is directly reported as a top-k result. The algorithm terminates if the top-k results have been found.

4.4.2 *Computing maximum scores*

Given a query Q and a node N in the R-tree, we compute a metric $maxf(N, Q)$, which offers an upper bound on the actual scores of the objects enclosed by N with respect to Q. That is,

$$maxf(N, Q) = \max\{f(O, Q) | O \text{ is enclosed by } N\} \quad (2)$$

Algorithm 1 Search(index R, GLRQ Q)

```
1: $TopK \leftarrow \emptyset$
2: $Queue$.push($R.root$, 0)
3: while $Queue \neq \emptyset$ do
4:     $(N, d) \leftarrow Queue$.pop()
5:     if $N$ is an object then
6:         $TopK$.insert($N$)
7:         if $|TopK| = k$ then
8:             break
9:         end if
10:    else if $N$ is not a leaf node then
11:        for entry $e$ in $N$ do
12:            $d \leftarrow maxf(e, Q)$
13:            if $d > 0$ then
14:                $Queue$.push($e, d$)
15:            end if
16:        end for
17:    else
18:        for entry $e$ in $N$ do
19:            $d \leftarrow f(e, Q)$
20:            if $d > 0$ then
21:                $Queue$.push($e, d$)
22:            end if
23:        end for
24:    end if
25: end while
26: return $TopK$
```

According to formula 1, the formula can be rewritten as follows.

$$maxf(N,Q) = \begin{cases} 0 & Q(N) = \emptyset \\ \max\limits_{O \in Q(N)} (g(prox(O,Q),\ rel(O,Q), \\ \quad s_1(O.A_1), \cdots)) & \text{otherwise} \end{cases} \tag{3}$$

where $Q(N)$ denotes the set of objects enclosed by N that satisfy the query predicate.

Therefore, to estimate $maxf(N, Q)$, we need first estimate whether $Q(N)$ is empty or not. Estimating $Q(N)$ is in fact estimating the satisfiability of a query, which is a special case of selectivity estimation. When $Q(N)$ is not empty, the value of $maxf(N, Q)$ can be estimated using the following formula:

$$maxf(N,Q) = g(prox(N,Q),\ rel(N,Q),\ s_1(N.A_1), \cdots) \tag{4}$$

where $prox(N, Q)$ is the proximity between $N.rect$ and $Q.\lambda$, $rel(N, Q)$ is the text similarity between $N.D$ and $Q.W$, and $s_i(N.A_i)$ is the score computed from the values of A_i in the dataset $N.D$.

4.4.3 Estimation

Given a node N in R-tree, estimating $Q(N)$ is not difficult using the synopsis kept for N. However, as the algorithm needs to visit a number of nodes, estimating for these nodes independently may take a long time. Considering the characteristics of the index structure, we devise a more efficient estimation method.

Conceptually, we can think of the synopsis associated with a node N as a multi-dimensional space (called *data space* of N, denoted as $ds(N)$) consisting of hyper-rectangles. Similarly, a query Q can be considered as a set of hyper-rectangles (called *query space* of Q, denoted as $qs(Q)$) encompassing the points satisfying the query predicate. Given a node N and a query Q, if $ds(N)$ intersects with $qs(Q)$, then Q is assumed to be satisfied at N, and we call $ds(N) \cap qs(Q)$ the *relevant data space (RDS)* of N with respect to Q, denoted as $rds_Q(N)$ (we simply use $rds(N)$ when the context is clear).

Our algorithm is described in Algorithm 2. It has two parts. In the first part, the RDS of the root $root$ of the synopses tree is precomputed. The algorithm traverses the junction tree in a leaf-to-root manner, and constructs the $rds(root)$ incrementally. Note that each node in the junction tree represents a clique, containing a pair of attributes. For each clique C, a partial RDS $rds_C(root)$ is computed (line 5). Here "partial" refers to the fact that only two attributes are concerned here. Then it merges the partial RDS with $rds(root)$ (line 6), which will add at least one dimension to $rds(root)$. Initially, $rds(root)$ is empty, so the merging operation just assigns $rds_C(root)$ to $rds(root)$. If $rds(root)$ is not empty, the merging will add a new dimension to $rds(root)$. This is because one attribute in C should have been seen in previous cliques, only the unseen attribute will be added into $rds(root)$. After that, the dimensions that will not be used are removed to reduce the dimensionality of $rds(root)$.

The second part is invoked when visiting an entry. Given an entry e, it first dynamically computes $rds(e)$. Let $par(e)$ be the parent of e. It can be inferred that, $rds(e)$ can be obtained by intersecting $ds(e)$ with $rds(par(e))$. Therefore, $rds(par(e))$ can be reused and estimation at e can be accelerated. Line 2 to 8 computes $rds(e)$ according to the discussion above. Then, based on $rds(e)$, the satisfiability of the query is estimated. If $rds(e)$ is empty, it implies that the query predicate cannot be satisfied by the objects enclosed by e. Otherwise, the algorithm estimates $maxf(e, Q)$ based on the stored keyword information and synopses according to formula 4.

We use a concrete example to illustrate the process.

Example 8: Recall the dataset in Fig. 2. The example 2D histograms H_1 and H_2 have been presented in Example 5. As the dataset is tiny, we assume that the buckets are not split any more. So each entry in the R-tree (see Fig. 6) just keeps 8 bits indicating whether there are descendant objects falling in the buckets. Consider the query in Example 3. Recall that the query predicate is "rating $\geq 4.0 \wedge$ health ≥ 95".

Algorithm 2 Estimation

```
Precompute(Junction tree $J$, global histograms $H$, query $Q$)
1:  let $root$ be the root of the synopses tree.
2:  $rds(root) \leftarrow \emptyset$
3:  for clique $C$ in $J$ do
4:                              ▷ traverse $J$ in a bottom-up way
5:      compute the $rds_C(root)$
6:      $rds(root) = merge(rds_C(root), rds(root))$
7:      reduce $rds(root)$ by removing the useless dimensions
8:  end for
Estimate $maxf(e, Q)$
1:  Let $par(e)$ be the parent of $e$
2:  $rds(e) \leftarrow rds(par(e))$
3:  for clique $C$ in $J$ do
4:      $rds(e) = rds(e) \cap ds_C(e)$
5:      if $rds(e) = \emptyset$ then
6:          break
7:      end if
8:  end for
9:  if $rds(e) = \emptyset$ then
10:     $maxf(e, Q) \leftarrow 0$
11: else
12:     compute $maxf(e, Q)$
13:                             ▷ compute based on formula 4
14: end if
15: return $maxf(e, Q)$
```

The junction tree has been constructed in Fig. 3. Let C_1 and C_2 refer to the cliques of {#reviews,health} and {#reviews,rating}, respectively. In the precomputation step, $rds(root)$ is computed. We can see from Fig. 7 that $rds(root)$ contains 6 hyper-rectangles $U_1, \cdots, U_6$. Note that the space is actually a plane ($92 \leq$ health ≤ 100). Consider the entry R_1. According to the information in Fig. 4, we can get its data space $ds(R_1)$ from the non-empty buckets in H_1 and H_2. As U_5 and U_6 intersect with $ds(R_1)$, $rds(R_1)$ is $\{U_5, U_6\}$. Now consider the entry R_4. As $rds(R_1)$ does not intersect with $ds_{C_1}(R_4)$, $rds(R_4)$ is empty, thus R_4 can be pruned.

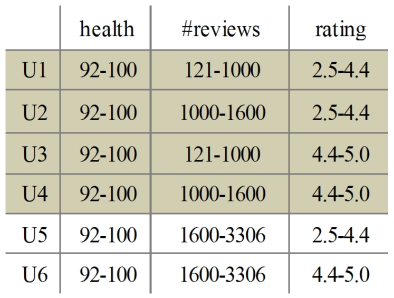
Note that Algorithm 2 as well as Example 8 just illustrates the main ideas. In implementation, we do not actually construct and maintain the hyper-rectangles.

An important feature of the metric $maxf(N, Q)$ is that it inherits the nice features of the R-tree for query processing [14].

Theorem 1: Given a query Q and a node N where N encloses a set $\mathcal{O}'$ of objects, we have $\forall O \in \mathcal{O}'$ $(maxf(N, Q) \geq f(O, Q))$.

Theorem 1 guarantees that the metric $maxf(N, Q)$ can be safely used to direct the traversal of the index tree. Due to space constraint, the proof of Theorem 1 is presented in Appendix B.

For some subsets of GLRQs, the algorithms can be



	health	#reviews	rating
U1	92-100	121-1000	2.5-4.4
U2	92-100	1000-1600	2.5-4.4
U3	92-100	121-1000	4.4-5.0
U4	92-100	1000-1600	4.4-5.0
U5	92-100	1600-3306	2.5-4.4
U6	92-100	1600-3306	4.4-5.0

Fig. 7. An example of data space

simpler. For example, for LKQs, the query processing algorithm behaves the same as IR-tree. If no keywords are involved in the query, inverted files are not needed. These algorithms are not presented to save space.

4.5 Maintenance of Indexes

The indexes in LINQ can be implemented within or outside a database. The core data structures we need are just R-trees and B$^+$-trees, which are available in mainstream databases. The maintenance of the IR-tree has been discussed in [3]. Therefore, we focus on the synopses tree here. We assume the domains of numeric attributes do not change and are known in advance.

Construct. In LINQ, the synopses are stored separately from the R-tree, therefore, the R-tree can be constructed as usual. Then, the global histograms are built, followed by the synopses (bit strings) of the nodes. The bit strings have the salient feature that, $N.syn$ can be obtained by superimposing $N_1.syn, \cdots, N_i.syn$, where $N_1, \cdots, N_i$ are the children of N. This feature can be used to speedup the construction of synopses, since only bit strings of leaf nodes need to be constructed from the scratch.

Insert. Insertion to the synopses tree is described in Algorithm 3, which is adapted from the standard algorithm in R-tree [15]. At line 3, the synopsis of N is updated, i.e. some bits are changed from 0s to 1s. The update is trivial as the new data $O.D$ only influences several partitions. When the leaf N is split into two ones N_1 and N_2 (line 6), the $N_1.syn$ and $N_2.syn$ are just computed from their entries. The update at line 10 and line 13 are similar. As the updates are propagated in a bottom-up way, the synopses of nodes at a higher level can be obtained by just superimposing the synopses of their children.

Delete. When some objects are deleted, the synopses of affected leafs have to be recomputed. If the deletion leads to reinsertion or merge of nodes, the affected internal nodes have to be updated. Again, update of internal nodes is done by superimposing the synopses of their children. Due to space limitation, the algorithm is not presented.

Algorithm 3 Insert(O)

```
1: $N \leftarrow$ chooseLeaf($O.\lambda$)
2: Add $O.\lambda$ to $N$
3: Update $N.syn$
4: if $N$ needs to be split then
5:     $\{N_1, N_2\} \leftarrow N$.split()
6:     Compute $N_1.syn$ and $N_2.syn$
7:     if $N$ is root then
8:         Initialize a new node $N_0$, let $N_1$ and $N_2$ be children of $N_0$, and set $N_0$ be the new root
9:     else
10:        Ascend from $N$ to the root, adjusting the covering MBRs, updating synopses and propagating node splits as necessary
11:    end if
12: else if $N$ is not root then
13:     Update the covering MBRs and synopses of the ancestors of $N$
14: end if
```

5 Experimental Study

In this section, we evaluate the performance of the proposed method.

5.1 Experimental Setting

Algorithms. The following methods are evaluated in the experiments: LKQ-naive, LKQ-F, PF, RT and LINQ, where the former four methods have been introduced in Section 3, and LINQ denotes our method. To make RT also support keyword search, each node in the R-tree is also associated with an inverted file as in IR-tree [2].

Data. We use both real and synthetic data for the experiments.

The real dataset is the Amazon product reviews dataset [16], which consists of product reviews obtained from www.amazon.com. In our experiments, we use the data describing members and their reviews. After cleaning, the dataset has about 5.8M objects and 5 attributes in addition to the geo-location and text attribute. Each object in the dataset represents a review, described by rating, number of feedback, number of helpful feedback, review body, consumer's location, rank, number of reviews, and so on. The review body is a piece of text whose length (number of keywords) ranges from 10 to 3345. We can think of an application where one tries to find some consumers according to their locations, reviews, and other information, thus the GLRQ can be adopted.

In the synthetic dataset, each object is composed of 9 attributes, including the coordinates, the text description, and 6 numeric attributes. The size of the synthetic dataset varies in the experiments, and will be pointed out later. The coordinates are randomly generated in (0, 100), and the texts are randomly picked from a Twitter dataset crawled by us. The average number of words per object is 7.4. The values of each attribute are randomly and independently generated, following a normal distribution. The domains and cardinalities of the attributes are different.

Setup. All algorithms were implemented in C++. The objects are stored in a B$^+$-tree file. We use the same settings as in IR-tree [2]. All experiments are conducted on a machine with i7 3.4G CPU, 4G main memory and 500G hard disk, running Windows 7. All indexes are disk-resident. When constructing the histograms, the bucketing scheme is *maxdiff* [17], which has shown good accuracy [9]. A B$^+$-tree is built on each numeric attribute, so that PF can utilize the indexes to accelerate the predicate-based filtering. The metrics considered are processing time and I/O cost, where the latter is measured by the number of disk page accesses.

The experimental settings are shown in Table 2. Some parameters have more than one value, which is used when studying the impacts of the parameters. The default values are used unless otherwise specified.

The ranking function in a GLRQ combines the spatio-textual similarity and scores of other attributes. We employ a linear combining function as in previous works [2], [5], like the one in Example 3. The weights of different parts may influence the performance. Since there are many weights in the ranking function, it is not easy to figure out the impact of a certain weight. Therefore, we simplify the ranking function as follows.

$$f(O,Q) = \alpha \times st\text{-}sim(O,Q) + (1-\alpha) \times att\text{-}score(O) \quad (5)$$

where $st\text{-}sim(O,Q)$ and $att\text{-}score(O)$ are the spatio-textual similarity and score derived from attribute values, respectively; they themselves are combinations of several factors. Then, we investigate the performance when the parameter α changes. For $st\text{-}sim(O,Q)$, we do not dig into the performance issues when the spatial proximity or text relevance has varying importance. That has been the topic of previous works [18]. We just treat different factors equally important. The $att\text{-}score(O)$ is treated similarly. To give an example, if the α parameter in Formula 5 is 0.6, then the actual function would be like this:

$$\begin{aligned} f(O,Q) =& 0.6 \times (0.5 \times prox(O,Q) + 0.5 \times rel(O,Q)) + \\ & 0.4 \times (0.5 \times \frac{O.rating}{5.0} + 0.5 \times \frac{O.health}{100}) \end{aligned}$$

5.2 Performance on Real Dataset

In this section, we report the experimental results on real dataset.

Index construction cost. We first evaluate the construction costs of various methods. The cost of an in-

TABLE 2
Experimental settings

Parameter	Values
Page size	4KB
R-tree fanout	100
Selectivity	0.01, 0.02, 0.05, **0.1**, 0.2, 0.5
Number of predicates	1, **2**, 3, 4, 5
Number of results	1, **5**, 10, 20, 35, 50
Number of buckets	512, 768, 1024, **1536**, 2048
Buffer capacity (%)	1, 2, **5**, 10, 20, 50
α	0.1, 0.3, 0.5, 0.7, **0.9**

(The default values are presented in bold)

dex is measured by its construction time and occupied space.

The costs of various methods are shown in Fig. 8, where *naive* refers to the LKQ-naive method. Note that we have to abort the indexing process of the naive method, because it takes so much time and space. We can see that, naive and LKQ-F show greatest index costs. The high index cost of LKQ-based methods has also been observed in other works [18], which makes them less attractive. PF has the lowest cost among the methods, because it requires only the availability of B$^+$-trees. RT and LINQ have moderate costs. The majority of their costs are due to the inverted files. Compared with RT, LINQ has smaller index size, but more indexing time. The time spent on building the synopses tree consists of two parts: the time of building global histograms, and the time of building synopses of nodes, where the former accounts for the vast majority. Note that the high cost of building histograms is not our fault, but a well-known problem [9]. And, the global histograms are not designed solely for our problem, it can be used by other components, e.g. the query optimizer, of the system. In fact, if we rip that part off, the additional cost of building the synopses of the nodes is very low. Still, the cost of building synopses can be reduced by using a simple bucketing strategy, such as equi-depth, and fewer buckets.

We also show the costs when the dataset has no text attributes in Fig. 8(c) and (d). It can be clearly seen that LINQ occupies the least space, but it has the greatest indexing time. The reason is similar.

In the following sections, we consider only four methods: LKQ-F, PF, LINQ and RT. We rule LKQ-naive out because it is deemed impractical, due to its huge space and time requirement.

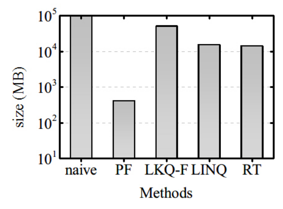
Processing time and disk I/O. We generate 50 distinct queries, involving different attributes. For each query, we fix the location and the keywords, and vary the selectivities of query predicates from 0.01 to 0.5 by changing the predicates. By this way, we obtain six query sets, corresponding to the six different selectivity levels. The processing times and I/O costs of the queries are averaged for each set.

We can see from Fig. 9 that, when the selectivity increases from 0.01 to 0.5, the processing time and disk I/O of PF increase, because PF is very sensitive to the predicate selectivity. The performance of RT fluctuates when the selectivity of query predicate changes, but there is no clear trend. For LKQ-F and LINQ, we can see that as the selectivity increases (i.e. the query predicate is less selective), the elapsed time and disk I/O cost go down. The reasons are different. For LKQ-F, that is because when the predicates are not selective, the results of the LKQs have higher probabilities of satisfying the predicates, thus less intermediate results are produced. For LINQ, the performance can be explained by false positives. False positives are introduced if some nodes of the R-tree are estimated to satisfy the predicate, but actually they do not. Obviously, false positives lead to unnecessary computations. When the predicates are not selective, the number of false positives decreases, so unnecessary computations are reduced.

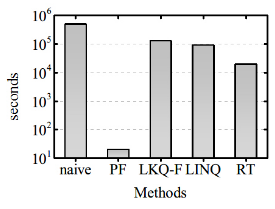
Varying the number of predicates. We change the number of selection predicates in the queries from 1 to 5. For each number of predicate, we generate a set of 50 queries. Each selection predicate is of the form $v_1 \leq attr \leq v_2$. The predicates in each query involve different attributes. The results are shown in Fig. 10. We can see that, when the number of predicates change, the time and I/O cost of LINQ have subtle changes. This is because when more predicates are present in the query, more attributes are involved, and estimating the satisfiability and scores is a bit more complicated. The disk I/O does not change much, because the synopsis of a node (the bit strings) is stored as a whole. The PF is very sensitive to the number of predicates, because more B$^+$-trees are searched when the number of predicates increases. The performance of LKQ-F does not change much, because its query performance is independent of the number of predicates.

Varying K. We examine the performances of various methods when the number K of requested results changes. We first generate 50 distinct queries. For each query, we vary K from 1 to 50. Then the results are averaged for each number of K. As seen from the Fig. 11, the cost of PF keeps almost constant, because the number of candidates satisfying the query does not change. Other methods have growing costs as K increases, but generally, the increase rate is not high. LINQ is always better than LKQ-F in terms of elapsed time and disk I/O, and the gap between their performance is enlarged when K increases. The performance can be explained that, when K is larger, LKQ-F needs to explore much more results, resulting in more disk page accesses.

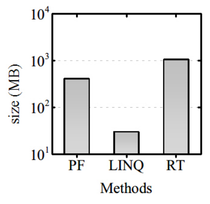
Varying the number of buckets. We change the number of buckets from 512 to 2048, and monitor 1)



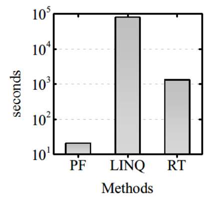
(a) Index size



(b) Indexing time

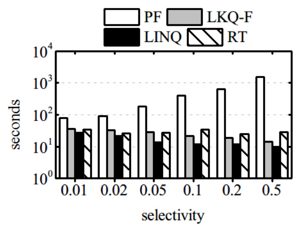


(c) Index size (without texts)

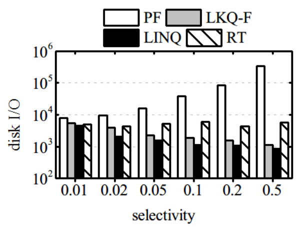


(d) Indexing time (without texts)

Fig. 8. Index construction costs

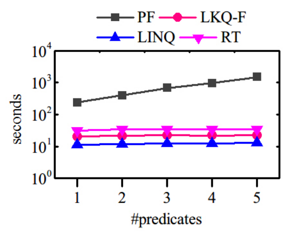


(a) Processing time

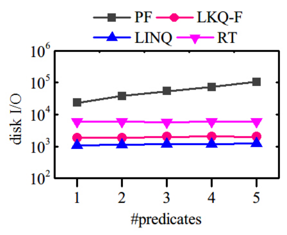


(b) Disk I/O

Fig. 9. Varying selectivities

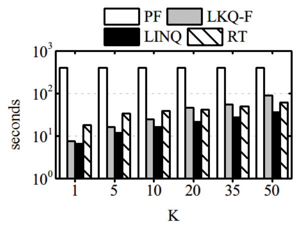


(a) Processing time

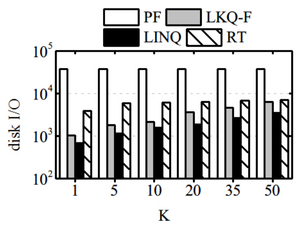


(b) Disk I/O

Fig. 10. Varying the number of predicates

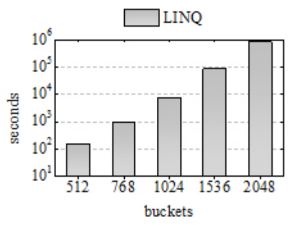


(a) Processing time

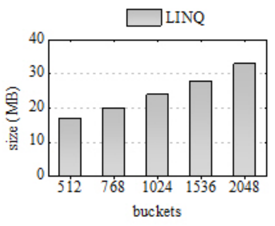


(b) Disk I/O

Fig. 11. Varying K



(a) Construction time



(b) Synopses size

Fig. 12. Construction costs vs. #bucket

the construction time and the size of the synopses, and 2) the query response time. Note that the number of buckets here refers to the number of partitions in Section 4, because the partitions here are equivalents of buckets in a general histogram.

Fig. 12 shows the construction time and size of the synopses. Not surprisingly, both increase with the number of buckets. The increase of time is much more evident than that of size. The high indexing time is mainly caused by the construction of multi-dimensional histograms, as explained earlier.

Fig. 13 shows the query processing time. We can see that the relationship between the query performance and number of buckets is intricate. When the number of buckets is small, the query performance gets worser. That is because the accuracy of the synopses is low at this time, and the pruning power of the synopses is weak. On the other hand, when the number of buckets increases, the query performance is not always better. The reason is twofold: 1) more time is spent on estimating the satisfiability of query predicate at each node; 2) as the synopses are compressed, more time is needed to do the uncompression.

Varying buffer size. We vary the buffer size from 1% to 20% of the index pages. Fig. 14 shows the I/O costs. The results of time do not show since the buffer size primarily influences the I/O cost. It can be seen that the disk I/Os decrease as the buffer size increases. This is because as the buffer size grows, more pages reside in the buffer, thus reducing the chances of reading data from the disk. The performance results are consistent with those of the previous sets of experiments.

Varying ranking function. We vary α to change the ranking function. We can see from Fig. 15 that the performance of PF is independent of the ranking function. LINQ and RT have some small fluctuations. LKQ-F is most sensitive to α in these methods. When α decreases, the performance of LKQ-F deteriorate. That is because the spatio-textual similarity accounts for a small portion of the final score, so it has to generate and examine more intermediate results.

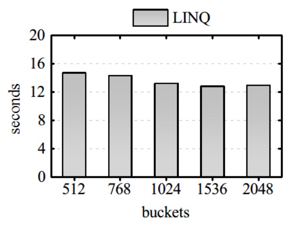


Fig. 13. Varying bucket number

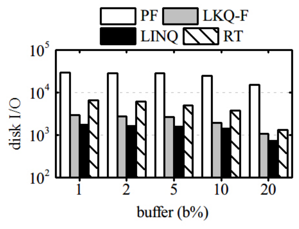
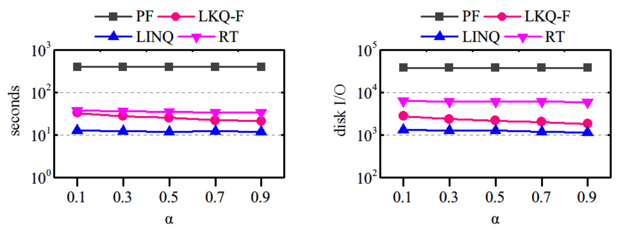


Fig. 14. Varying buffer size



(a) Processing time (b) Disk I/O

Fig. 15. Varying parameter α

5.3 Performance on Synthetic Datasets

Varying the size of dataset. In this set of experiments, we show how query performance varies with dataset size. We increase the number of objects in the synthetic dataset from 2M to 10M. It can be seen from Fig. 16 that, PF has more obvious increases in time and disk I/O when the data size increases, which can be explained by its frequent disk accesses. On the other hand, the LINQ, LKQ-F and RT are more stable.

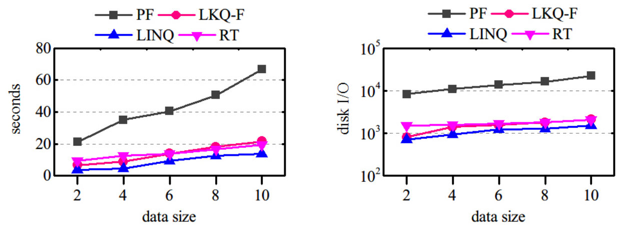
We also show the costs of synopses construction on datasets with different sizes. As can be seen from Fig. 17, the construction time grows with the dataset, but it scales well. The space occupied by the synopses has insignificant increase, because the size of the synopses is mainly dependent on the number of buckets.

Varying the dimensionality We study the query performance of the methods when the number of attributes increases from 2 to 10. We only show the disk I/O of the methods, which are depicted in Fig. 18. The cost of PF is orders of magnitude higher than other methods, thus is not shown to avoid cluttering the figure.

We can see that the disk I/O of LINQ increases at a slow rate. This is because when the dimensionality is high, the estimation error rate goes up, and more leaf nodes are visited. As for RT, its performance degenerates seriously when the dimensionality is high, which is in consistent with the observations in other works (e.g. [19]). LKQ-F has a steady cost when the dimensionality increases. From this set of experiments, we can see that, our method is appealing when the dataset has a low or moderate number of dimensions, which is typical of many real life applications. The problem LINQ encounters when the dimensionality is high can be largely explained by *the curse of dimensionality*, a well-known problem faced by any techniques processing high-dimensional data. To process GLRQ on high-dimensional data, we can choose to simplify the synopses (e.g. use Value Independancy Assumption), or switch to some simple method, such as LKQ-F.

6 Related Work

Location-aware rank query (LRQ) is an important class of query, and several subsets have been studied.



(a) Processing time (b) Disk I/O

Fig. 16. Varying data size

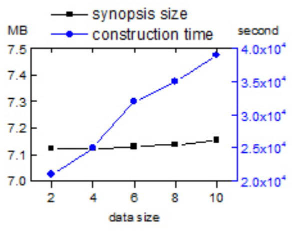


Fig. 17. Indexing cost on synthetic datasets

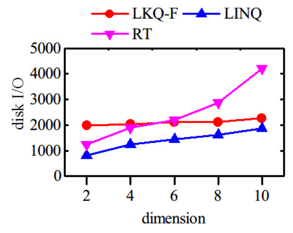


Fig. 18. Varying dimensionality

Nearest neighbor (NN) query is the most well-known LRQ. It takes a spatial location as input and outputs the closest objects in the dataset. The existing algorithms use index structures (most often an R-tree [15]) and some pruning methods to restrict the search space. Several NN algorithms have been proposed, where the *best-first* algorithm of Hjaltason and Samet [14] is one of the most influential methods, which proves to achieve the optimal I/O performance.

Location-aware keyword query (LKQ, a.k.a spatial keyword query) is another class of LRQ. The basic LKQ does not contain any predicates, and has a ranking function combining spatial proximity and textual relevance. There are also some other variants of LKQs. One variant specifies a spatial region [20], which restricts the locations of the results. Another variant interprets the keywords as boolean conditions [1]. That is, it retrieves the objects containing all the keywords, and ranks the results just according to the spatial distances. All these query types are covered by the GLRQ formulated in this work. The basic LKQ is more fundamental, and thus has been researched extensively by [1], [2], [3], [4], [5] and [6]. Readers are referred to the survey [21] for more details about LKQ processing. The query we studied in this paper is more generic than LKQ. Although some existing LKQ techniques can be extended to process GLRQ, the solutions are not efficient.

The location optimization is an important problem related to location-aware query. The typical setting of the problem is as follows. Given a set of clients and some facilities, the problem finds certain location(s) such that if new facilities are established at the location(s), a certain optimization function is minimized [22], [23]. The problem is related to NN and RNN (reverse nearest neighbor) search. But the solutions does not touch the core problem of summarizing numeric attributes and estimating in GLRQ, thus do not apply.

We propose the synopses tree to summarize the numeric attributes. Some data structures have been proposed for similar purpose. In aR-tree [13], each entry in an R-tree is associated with an aggregated value, e.g. the count of objects or sum of the associated values. In [24], each entry in an R-tree node is associated with a numeric range and a trie, in order to support mixed-type fuzzy queries with conditions on numeric and string attributes. Our synopses tree stores synopses instead of simple aggregate values, thus is more challenging.

7 Conclusions

In this paper, we formulate an important class of query, generic location-aware rank query (GLRQ), and propose a framework called LINQ to process the query. In this framework, a novel index structure called synopses tree is built, which indexes synopses of objects, and enables efficient pruning and estimation. The synopses tree is designed to reduce the cost of construction while preserving accuracy. We show how to process GLRQs efficiently in the LINQ framework, leveraging synopses tree and other index structures. Experimental results show that the proposed framework is effective and efficient.

References

[1] I. De Felipe, V. Hristidis, and N. Rishe, "Keyword search on spatial databases," in *Proc. Int'l Conf. Data Eng. (ICDE)*, 2008, pp. 656–665.

[2] G. Cong, C. S. Jensen, and D. Wu, "Efficient retrieval of the top-k most relevant spatial web objects," *Proc. VLDB Endow.*, vol. 2, pp. 337–348, 2009.

[3] D. Wu, G. Cong, and C. Jensen, "A framework for efficient spatial web object retrieval," *The VLDB Journal*, pp. 1–26, 2012.

[4] Z. Li, K. Lee, B. Zheng, W.-C. Lee, D. L. Lee, and X. Wang, "IR-Tree: an efficient index for geographic document search," *IEEE Trans. Knowledge and Data Eng.*, vol. 23, no. 4, pp. 585–599, 2011.

[5] J. a. B. Rocha-Junior, O. Gkorgkas, S. Jonassen, and K. Nørvåg, "Efficient processing of top-k spatial keyword queries," in *Proc. Int'l Conf Advances in spatial and temporal databases (SSTD)*, 2011, pp. 205–222.

[6] Chengyuan Zhang, Ying Zhang, Wenjie Zhang, and Xuemin Lin, "Inverted linear quadtree : Efficient top k spatial keyword search," in *Proc. Int'l Conf. Data Eng. (ICDE)*, 2013.

[7] A. Silberschatz, H. F. Korth, and S. Sudarshan, *Database system concepts*. McGraw-Hill Higher Education, 2006.

[8] K. S. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft, "When is nearest neighbor meaningful?" in *Proc. Int'l Conf. Database Theory (ICDT)*, 1999, pp. 217–235.

[9] G. Cormode, M. Garofalakis, P. J. Haas, and C. Jermaine, "Synopses for massive data: Samples, histograms, wavelets, sketches," *Found. Trends databases*, vol. 4, no. 1–3, pp. 1–294, 2012.

[10] K. Tzoumas, A. Deshpande, and C. S. Jensen, "Lightweight graphical models for selectivity estimation without independence assumptions," *Proc. VLDB Endow.*, vol. 4, no. 11, pp. 852–863, 2011.

[11] F. Buccafurri, G. Lax, S. Domenico, L. Pontieri, and D. Rosaci, "Enhancing histograms by tree-like bucket indices," *The VLDB Journal*, vol. 17, no. 5, pp. 1041–1061, 2008.

[12] D. Lemire, O. Kaser, and K. Aouiche, "Sorting improves word-aligned bitmap indexes," *Data Knowl. Eng.*, vol. 69, no. 1, pp. 3–28, 2010.

[13] D. Papadias, P. Kalnis, J. Zhang, and Y. Tao, "Efficient OLAP operations in spatial data warehouses," in *Proc. Int'l Symp. Advances in Spatial and Temporal Databases (SSTD)*, 2001, pp. 443–459.

[14] G. R. Hjaltason and H. Samet, "Distance browsing in spatial databases," *ACM Trans. Database Syst.*, vol. 24, no. 2, pp. 265–318, 1999.

[15] A. Guttman, "R-trees: a dynamic index structure for spatial searching," *SIGMOD Rec.*, vol. 14, no. 2, pp. 47–57, 1984.

[16] N. Jindal and B. Liu, "Opinion spam and analysis," in *Proc. Int'l Conf. Web Search and Data Mining (WSDM)*, 2008, pp. 219–230.

[17] V. Poosala, P. J. Haas, Y. E. Ioannidis, and E. J. Shekita, "Improved histograms for selectivity estimation of range predicates," *SIGMOD Rec.*, vol. 25, no. 2, pp. 294–305, 1996.

[18] L. Chen, G. Cong, C. S. Jensen, and D. Wu, "Spatial keyword query processing: an experimental evaluation," in *Proc. VLDB Endow.*, 2013, pp. 217–228.

[19] S. Berchtold, D. A. Keim, and H.-P. Kriegel, "The x-tree: An index structure for high-dimensional data," in *Proc. Int'l Conf. Very Large Data Bases (VLDB)*, 1996, pp. 28–39.

[20] J. Fan, G. Li, L. Zhou, S. Chen, and J. Hu, "Seal: spatio-textual similarity search," *Proc. VLDB Endow.*, vol. 5, no. 9, pp. 824–835, 2012.

[21] X. Cao, L. Chen, G. Cong, C. S. Jensen, Q. Qu, A. Skovsgaard, D. Wu, and M. L. Yiu, "Spatial keyword querying," in *Proc. Int'l Conf. Conceptual Modeling (ER)*, 2012, pp. 16–29.

[22] J. Qi, R. Zhang, L. Kulik, D. Lin, and Y. Xue, "The min-dist location selection query," in *Proc. Int'l Conf. Data Eng. (ICDE)*, 2012, pp. 366–377.

[23] Y. Sun, J. Huang, Y. Chen, R. Zhang, and X. Du, "Location selection for utility maximization with capacity constraints," in *Proc. Int'l Conf. Information and Knowledge Management (CIKM)*, 2012, pp. 2154–2158.

[24] L. Jin, C. Li, N. Koudas, and A. K. H. Tung, "Indexing mixed types for approximate retrieval," in *Proc. Int'l Conf. Very Large Data Bases (VLDB)*, 2005, pp. 793–804.

Xiping Liu is a lecturer in Jiangxi University of Finance and Economics, China. His research interests include keyword search over database and XML data management.

Lei Chen is an associate professor in Hong Kong University of Science and Technology. His research interests include spatial database, graph databases and time series databases. He is a member of the IEEE.

Changxuan Wan is a full professor in Jiangxi University of Finance and Economics. His research interests include Web data management, data mining and information retrieval.