Fast Nearest Neighbor Search with Keywords

Yufei Tao and Cheng Sheng

Abstract—Conventional spatial queries, such as range search and nearest neighbor retrieval, involve only conditions on objects' geometric properties. Today, many modern applications call for novel forms of queries that aim to find objects satisfying both a spatial predicate, and a predicate on their associated texts. For example, instead of considering all the restaurants, a nearest neighbor query would instead ask for the restaurant that is the closest among those whose menus contain "steak, spaghetti, brandy" all at the same time. Currently, the best solution to such queries is based on *the IR^2-tree*, which, as shown in this paper, has a few deficiencies that seriously impact its efficiency. Motivated by this, we develop a new access method called *the spatial inverted index* that extends the conventional inverted index to cope with multidimensional data, and comes with algorithms that can answer nearest neighbor queries with keywords in real time. As verified by experiments, the proposed techniques outperform the IR^2-tree in query response time significantly, often by a factor of orders of magnitude.

Index Terms—Nearest neighbor search, keyword search, spatial index

1 Introduction

A spatial database manages multidimensional objects (such as points, rectangles, etc.), and provides fast access to those objects based on different selection criteria. The importance of spatial databases is reflected by the convenience of modeling entities of reality in a geometric manner. For example, locations of restaurants, hotels, hospitals and so on are often represented as points in a map, while larger extents such as parks, lakes, and landscapes often as a combination of rectangles. Many functionalities of a spatial database are useful in various ways in specific contexts. For instance, in a geography information system, range search can be deployed to find all restaurants in a certain area, while nearest neighbor retrieval can discover the restaurant closest to a given address.

Today, the widespread use of search engines has made it realistic to write spatial queries in a brandnew way. Conventionally, queries focus on objects' geometric properties only, such as whether a point is in a rectangle, or how close two points are from each other. We have seen some modern applications that call for the ability to select objects based on *both* of their geometric coordinates and their associated texts. For example, it would be fairly useful if a search engine can be used to find the nearest restaurant that offers "steak, spaghetti, and brandy" all at the same time. Note that this is not the "globally" nearest restaurant (which would have been returned by a traditional nearest neighbor query), but the nearest restaurant among *only* those providing all the demanded foods and drinks.

There are easy ways to support queries that combine spatial and text features. For example, for the above query, we could first fetch all the restaurants whose menus contain the set of keywords {steak, spaghetti, brandy}, and then from the retrieved restaurants, find the nearest one. Similarly, one could also do it reversely by targeting first the spatial conditions—browse all the restaurants in ascending order of their distances to the query point until encountering one whose menu has all the keywords. The major drawback of these straightforward approaches is that they will fail to provide real time answers on difficult inputs. A typical example is that the real nearest neighbor lies quite faraway from the query point, while all the closer neighbors are missing at least one of the query keywords.

Spatial queries with keywords have not been extensively explored. In the past years, the community has sparked enthusiasm in studying keyword search in relational databases. It is until recently that attention was diverted to multidimensional data [12], [13], [21]. The best method to date for nearest neighbor search with keywords is due to Felipe et al. [12]. They nicely integrate two well-known concepts: *R-tree* [2], a popular spatial index, and *signature file* [11], an effective method for keyword-based document retrieval. By doing so they develop a structure called the *IR^2-tree* [12], which has the strengths of both R-trees and signature files. Like R-trees, the IR^2-tree preserves objects' spatial proximity, which is the key to solving spatial queries efficiently. On the other hand, like signature files, the IR^2-tree is able to filter a considerable portion of the objects that do not contain all the query keywords, thus significantly reducing the number of objects to be examined.

The IR^2-tree, however, also inherits a drawback of signature files: *false hits*. That is, a signature file, due to its conservative nature, may still direct the search to some objects, even though they do not have all the keywords. The penalty thus caused is the need to *verify* an object whose satisfying a query or not cannot be resolved using only its signature, but requires loading its full text

- *Y. Tao is with the Department of Computer Science and Engineering, Chinese University of Hong Kong, Hong Kong, and with the Division of Web Science and Technology, Korea Advanced Institute of Science and Technology, Korea. E-mail: taoyf@cse.cuhk.edu.hk.*
- *C. Sheng is with Google Switzerland. E-mail: jeru.sheng@gmail.com.*

Manuscript received 4 Nov. 2012; revised 10 Mar. 2013; accepted 6 Apr. 2013; date of publication 25 Apr. 2013; date of current version 18 Mar. 2014.
Recommended for acceptance by X. Zhou.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TKDE.2013.66

description, which is expensive due to the resulting random accesses. It is noteworthy that the false hit problem is not specific only to signature files, but also exists in other methods for approximate set membership tests with compact storage (see [7] and the references therein). Therefore, the problem cannot be remedied by simply replacing signature file with any of those methods.

In this paper, we design a variant of inverted index that is optimized for multidimensional points, and is thus named the *spatial inverted index* (SI-index). This access method successfully incorporates point coordinates into a conventional inverted index with small extra space, owing to a delicate compact storage scheme. Meanwhile, an SI-index preserves the spatial locality of data points, and comes with an R-tree built on every inverted list at little space overhead. As a result, it offers two competing ways for query processing. We can (sequentially) merge multiple lists very much like merging traditional inverted lists by ids. Alternatively, we can also leverage the R-trees to browse the points of all relevant lists in ascending order of their distances to the query point. As demonstrated by experiments, the SI-index significantly outperforms the IR^2-tree in query efficiency, often by a factor of orders of magnitude.

The rest of the paper is organized as follows. Section 2 defines the problem studied in this paper formally. Section 3 surveys the previous work related to ours. Section 4 gives an analysis that reveals the drawbacks of the IR-tree. Section 5 presents a distance browsing algorithm for performing keyword-based nearest neighbor search. Section 6 proposes the SI-idnex, and establishes its theoretical properties. Section 7 evaluates our techniques with extensive experiments. Section 8 concludes the paper with a summary of our findings.

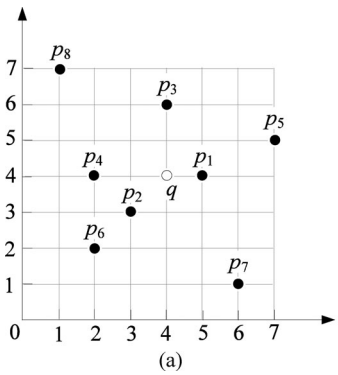
2 Problem Definitions

Let P be a set of multidimensional points. As our goal is to combine keyword search with the existing location-finding services on facilities such as hospitals, restaurants, hotels, etc., we will focus on dimensionality 2, but our technique can be extended to arbitrary dimensionalities with no technical obstacle. We will assume that the points in P have integer coordinates, such that each coordinate ranges in $[0, t]$, where t is a large integer. This is not as restrictive as it may seem, because even if one would like to insist on real-valued coordinates, the set of different coordinates representable under a space limit is still finite and enumerable; therefore, we could as well convert everything to integers with proper scaling.

As with [12], each point $p \in P$ is associated with a set of words, which is denoted as W_p and termed the *document* of p. For example, if p stands for a restaurant, W_p can be its menu, or if p is a hotel, W_p can be the description of its services and facilities, or if p is a hospital, W_p can be the list of its out-patient specialities. It is clear that W_p may potentially contain numerous words.

Traditional nearest neighbor search returns the data point closest to a query point. Following [12], we extend the problem to include predicates on objects' texts. Formally, in our context, a *nearest neighbor* (NN) *query* specifies a point q and a set W_q of keywords (we refer to W_q as the *document* of the query). It returns the point in P_q that is the nearest to q, where P_q is defined as

$$P_q = \{p \in P \mid W_q \subseteq W_p\}. \quad (1)$$



p	W_p
p_1	{a, b}
p_2	{b, d}
p_3	{d}
p_4	{a, e}
p_5	{c, e}
p_6	{c, d, e}
p_7	{b, e}
p_8	{c, d}

(b)

Fig. 1. (a) Shows the locations of points and (b) gives their associated texts.

In other words, P_q is the set of objects in P whose documents contain all the keywords in W_q. In the case where P_q is empty, the query returns nothing. The problem definition can be generalized to k nearest neighbor (kNN) search, which finds the k points in P_q closest to q; if P_q has less than k points, the entire P_q should be returned.

For example, assume that P consists of eight points whose locations are as shown in Fig. 1a (the black dots), and their documents are given in Fig. 1b. Consider a query point q at the white dot of Fig. 1a with the set of keywords $W_q = \{c, d\}$. Nearest neighbor search finds p_6, noticing that all points closer to q than p_6 are missing either the query keyword c or d. If $k = 2$ nearest neighbors are wanted, p_8 is also returned in addition. The result is still $\{p_6, p_8\}$ even if k increases to 3 or higher, because only two objects have the keywords c and d at the same time.

We consider that the data set does not fit in memory, and needs to be indexed by efficient access methods in order to minimize the number of I/Os in answering a query.

3 Related Work

Section 3.1 reviews the *information retrieval R-tree* (IR^2-tree) [12], which is the state of the art for answering the nearest neighbor queries defined in Section 2. Section 3.2 explains an alternative solution based on the inverted index. Finally, Section 3.3 discusses other relevant work in spatial keyword search.

3.1 The IR^2-Tree

As mentioned before, the IR^2-tree [12] combines the R-tree with signature files. Next, we will review what is a signature file before explaining the details of IR^2-trees. Our discussion assumes the knowledge of R-trees and the *best-first* algorithm [14] for NN search, both of which are well-known techniques in spatial databases.

Signature file in general refers to a hashing-based framework, whose instantiation in [12] is known as *superimposed coding* (SC), which is shown to be more effective than other

word	*hashed bit string*
a	00101
b	01001
c	00011
d	00110
e	10010

Fig. 2. Example of bit string computation with $l = 5$ and $m = 2$.

instantiations [11]. It is designed to perform *membership tests*: determine whether a query word w exists in a set W of words. SC is conservative, in the sense that if it says "no", then w is definitely not in W. If, on the other hand, SC returns "yes", the true answer can be either way, in which case the whole W must be scanned to avoid a false hit.

In the context of [12], SC works in the same way as the classic technique of *bloom filter*. In preprocessing, it builds a bit signature of length l from W by hashing each word in W to a string of l bits, and then taking the disjunction of all bit strings. To illustrate, denote by $h(w)$ the bit string of a word w. First, all the l bits of $h(w)$ are initialized to 0. Then, SC repeats the following m times: randomly choose a bit and set it to 1. Very importantly, randomization must use w as its seed to ensure that the same w always ends up with an identical $h(w)$. Furthermore, the m choices are mutually independent, and may even happen to be the same bit. The concrete values of l and m affect the space cost and false hit probability, as will be discussed later.

Fig. 2 gives an example to illustrate the above process, assuming $l = 5$ and $m = 2$. For example, in the bit string $h(a)$ of a, the third and fifth (counting from left) bits are set to 1. As mentioned earlier, the bit signature of a set W of words simply ORs the bit strings of all the members of W. For instance, the signature of a set $\{a, b\}$ equals 01101, while that of $\{b, d\}$ equals 01111.

Given a query keyword w, SC performs the membership test in W by checking whether all the 1s of $h(w)$ appear at the same positions in the signature of W. If not, it is guaranteed that w cannot belong to W. Otherwise, the test cannot be resolved using only the signature, and a scan of W follows. A *false hit* occurs if the scan reveals that W actually does not contain w. For example, assume that we want to test whether word c is a member of set $\{a, b\}$ using only the set's signature 01101. Since the fourth bit of $h(c) = 00011$ is 1 but that of 01101 is 0, SC immediately reports "no". As another example, consider the membership test of c in $\{b, d\}$ whose signature is 01111. This time, SC returns "yes" because 01111 has 1s at all the bits where $h(c)$ is set to 1; as a result, a full scan of the set is required to verify that this is a false hit.

The IR^2-tree is an R-tree where each (leaf or nonleaf) entry E is augmented with a signature that summarizes the union of the texts of the objects in the subtree of E. Fig. 3 demonstrates an example based on the data set of Fig. 1 and the hash values in Fig. 2. The string 01111 in the leaf entry p_2, for example, is the signature of $W_{p_2} = \{b, d\}$ (which is the document of p_2; see Fig. 1b). The string 11111 in the nonleaf entry E_3 is the signature of $W_{p_2} \cup W_{p_6}$, namely, the set of all words describing p_2 and p_6. Notice that, in general, the signature of a nonleaf entry E can be conveniently obtained simply as the disjunction of all the signatures in the child node of E. A nonleaf signature may allow a query algorithm to realize that a certain word cannot exist in the subtree. For example, as the second bit of $h(b)$ is 1, we know that no object in the subtrees of E_4 and E_6 can have word b in its texts—notice that the signatures of E_4 and E_6 have 0 as their second bits. In general, the signatures in an IR^2-tree may have different lengths at various levels.

On conventional R-trees, the best-first algorithm [14] is a well-known solution to NN search. It is straightforward to adapt it to IR^2-trees. Specifically, given a query point q and a keyword set W_q, the adapted algorithm accesses the entries of an IR^2-tree in ascending order of the distances of their MBRs to q (the MBR of a leaf entry is just the point itself), pruning those entries whose signatures indicate the absence of at least one word of W_q in their subtrees. Whenever a leaf entry, say of point p, cannot be pruned, a random I/O is performed to retrieve its text description W_p. If W_q is a subset of W_p, the algorithm terminates with p as the answer; otherwise, it continues until no more entry remains to be processed. In Fig. 3, assume that the query point q has a keyword set $W_q = \{c, d\}$. It can be verified that the algorithm must read all the nodes of the tree, and fetch the documents of p_2, p_4, and p_6 (in this order). The final answer is p_6, while p_2 and p_4 are false hits.

3.2 Solutions Based on Inverted Indexes

Inverted indexes (I-index) have proved to be an effective access method for keyword-based document retrieval. In the spatial context, nothing prevents us from treating the

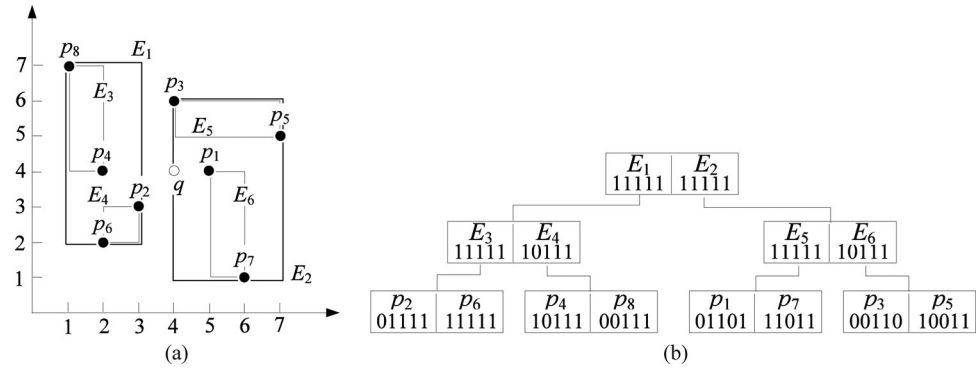


Fig. 3. Example of an IR^2-tree. (a) shows the MBRs of the underlying R-tree and (b) gives the signatures of the entries.

word	inverted list
a	p_1 p_4
b	p_1 p_2 p_7
c	p_5 p_6 p_8
d	p_2 p_3 p_6 p_8
e	p_4 p_5 p_6 p_7

Fig. 4. Example of an inverted index.

text description W_p of a point p as a document, and then, building an I-index. Fig. 4 illustrates the index for the data set of Fig. 1. Each word in the vocabulary has an inverted list, enumerating the ids of the points that have the word in their documents.

Note that the list of each word maintains a sorted order of point ids, which provides considerable convenience in query processing by allowing an efficient merge step. For example, assume that we want to find the points that have words c and d. This is essentially to compute the intersection of the two words' inverted lists. As both lists are sorted in the same order, we can do so by merging them, whose I/O and CPU times are both linear to the total length of the lists.

Recall that, in NN processing with IR^2-tree, a point retrieved from the index must be *verified* (i.e., having its text description loaded and checked). Verification is also necessary with I-index, but for exactly the opposite reason. For IR^2-tree, verification is because we do not have the detailed texts of a point, while for I-index, it is because we do not have the coordinates. Specifically, given an NN query q with keyword set W_q, the query algorithm of I-index first retrieves (by merging) the set P_q of all points that have all the keywords of W_q, and then, performs $|P_q|$ random I/Os to get the coordinates of each point in P_q in order to evaluate its distance to q.

According to the experiments of [12], when W_q has only a single word, the performance of I-index is very bad, which is expected because *everything* in the inverted list of that word must be verified. Interestingly, as the size of W_q increases, the performance gap between I-index and IR^2-tree keeps narrowing such that I-index even starts to outperform IR^2-tree at $|W_q| = 4$. This is not as surprising as it may seem. As $|W_q|$ grows large, not many objects need to be verified because the number of objects carrying all the query keywords drops rapidly. On the other hand, at this point an advantage of I-index starts to pay off. That is, scanning an inverted list is relatively cheap because it involves only sequential I/Os,[1] as opposed to the random nature of accessing the nodes of an IR^2-tree.

3.3 Other Relevant Work

Our treatment of nearest neighbor search falls in the general topic of *spatial keyword search*, which has also given rise to several alternative problems. A complete survey of all those problems goes beyond the scope of this paper. Below we mention several representatives, but interested readers can refer to [4] for a nice survey.

Cong et al. [10] considered a form of keyword-based nearest neighbor queries that is similar to our formulation, but differs in how objects' texts play a role in determining the query result. Specifically, aiming at an IR flavor, the approach of [10] computes the *relevance* between the documents of an object p and a query q. This relevance score is then integrated with the Euclidean distance between p and q to calculate an overall similarity of p to q. The few objects with the highest similarity are returned. In this way, an object may still be in the query result, even though its document does not contain all the query keywords. In our method, same as [12], object texts are utilized in evaluating a *boolean* predicate, i.e., if any query keyword is missing in an object's document, it must not be returned. Neither approach subsumes the other, and both make sense in different applications. As an application in our favor, consider the scenario where we want to find a close restaurant serving "steak, spaghetti and brandy", and do not accept any restaurant that do not serve any of these three items. In this case, a restaurant's document either fully satisfies our requirement, or does not satisfy at all. There is no "partial satisfaction", as is the rationale behind the approach of [10].

In *geographic web search*, each webpage is assigned a geographic region that is pertinent to the webpage's contents. In web search, such regions are taken into account so that higher rankings are given to the pages in the same area as the location of the computer issuing the query (as can be inferred from the computer's IP address) [8], [13], [21]. The underpinning problem that needs to be solved is different from keyword-based nearest neighbor search, but can be regarded as the combination of keyword search and *range queries*.

Zhang et al. [20] dealt with the so-called *m-closest keywords problem*. Specifically, let P be a set of points each of which carries a single keyword. Given a set W_q of query keywords (note: no query point q is needed), the goal is to find $m = |W_q|$ points from P such that (i) each point has a distinct keyword in W_q, and (ii) the maximum mutual distance of these points is minimized (among all subsets of m points in P fulfilling the previous condition). In other words, the problem has a "collaborative" nature in that the resulting m points should cover the query keywords together. This is fundamentally different from our work where there is no sense of collaboration at all, and instead the quality of each *individual* point with respect to a query can be quantified into a concrete value. Cao et al. [6] proposed *collective spatial keyword querying*, which is based on similar ideas, but aims at optimizing different objective functions.

In [5], Cong et al. proposed the concept of *prestige-based spatial keyword search*. The central idea is to evaluate the similarity of an object p to a query by taking also into account the objects in the neighborhood of p. Lu et al. [17] recently combined the notion of keyword search with *reverse nearest neighbor queries*.

Although keyword search has only started to receive attention in spatial databases, it is already thoroughly studied in relational databases, where the objective is to enable a querying interface that is similar to that of search engines, and can be easily used by naive users without knowledge

1. Strictly speaking, this is not precisely true because merging may need to jump across different lists; however, random I/Os will account for only a small fraction of the total overhead as long as a proper prefetching strategy is employed, e.g., reading 10 sequential pages at a time.

about SQL. Well known systems with such mechanisms include *DBXplorer* [1], *Discover* [15], *Banks* [3], and so on. Interested readers may refer to [9] for additional references into that literature.

4 Drawbacks of the IR^2-Tree

The IR^2-tree is the first access method for answering NN queries with keywords. As with many pioneering solutions, the IR^2-tree also has a few drawbacks that affect its efficiency. The most serious one of all is that the number of false hits can be really large when the object of the final result is faraway from the query point, or the result is simply empty. In these cases, the query algorithm would need to load the documents of many objects, incurring expensive overhead as each loading necessitates a random access.

To explain the details, we need to first discuss some properties of SC (the variant of signature file used in the IR^2-tree). Recall that, at first glance, SC has two parameters: the length l of a signature, and the number m of bits chosen to set to 1 in hashing a word. There is, in fact, really just a single parameter l, because the optimal m (which minimizes the probability of a false hit) has been solved by Stiassny [18]:

$$m_{opt} = l \cdot \ln(2)/g, \tag{2}$$

where g is the number of distinct words in the set W on which the signature is being created. Even with such an optimal choice of m, Faloutsos and Christodoulakis [11] show that the false hit probability equals

$$P_{false} = (1/2)^{m_{opt}}. \tag{3}$$

Put in a different way, given any word w that does *not* belong to W, SC will still report "yes" with probability P_{false}, and demand a full scan of W.

It is easy to see that P_{false} can be made smaller by adopting a larger l (note that g is fixed as it is decided by W). In particular, asymptotically speaking, to make sure P_{false} is at least a constant, l must be $\Omega(g)$, i.e., the signature should have $\Omega(1)$ bit for every distinct word of W. Indeed, for the IR^2-tree, Felipe et al. [12] adopt a value of l that is approximately equivalent to $4g$ in their experiments (g here is the average number of distinct words a data point has in its text description). It thus follows that

$$P_{false} = (1/2)^{4\ln(2)} = 0.15. \tag{4}$$

The above result takes a heavy toll on the efficiency of the IR^2-tree. For simplicity, let us first assume that the query keyword set W_q has only a single keyword w (i.e., $|W_q| = 1$). Without loss of generality, let p be the object of the query result, and S be the set of data points that are closer to the query point q than p. In other words, none of the points in S has w in their text documents (otherwise, p cannot have been the final result). By Equation (4), roughly 15 percent of the points in S *cannot* be pruned using their signatures, and thus, will become false hits. This also means that the NN algorithm is expected to perform at least $0.15|S|$ random I/Os.

So far we have considered $|W_q| = 1$, but the discussion extends to arbitrary $|W_q|$ in a straightforward manner. It is easy to observe (based on Equation (4)) that, in general, the false hit probability satisfies

$$P_{false} \geq 0.15^{|W_q|}. \tag{5}$$

When $|W_q| > 1$, there is another negative fact that adds to the deficiency of the IR^2-tree: for a greater $|W_q|$, the expected size of S increases dramatically, because fewer and fewer objects will contain all the query keywords. The effect is so severe that the number of random accesses, given by $P_{false}|S|$, may escalate as $|W_q|$ grows (even with the decrease of P_{false}). In fact, as long as $|W_q| > 1$, S can easily be the entire data set when the user tries out an uncommon combination of keywords that does not exist in any object. In this case, the number of random I/Os would be so prohibitive that the IR^2-tree would not be able to give real time responses.

5 Merging and Distance Browsing

Since verification is the performance bottleneck, we should try to avoid it. There is a simple way to do so in an I-index: one only needs to store the coordinates of each point together with each of its appearances in the inverted lists. The presence of coordinates in the inverted lists naturally motivates the creation of an R-tree on each list indexing the points therein (a structure reminiscent of the one in [21]). Next, we discuss how to perform keyword-based nearest neighbor search with such a combined structure.

The R-trees allow us to remedy an awkwardness in the way NN queries are processed with an I-index. Recall that, to answer a query, currently we have to first get all the points carrying all the query words in W_q by merging several lists (one for each word in W_q). This appears to be unreasonable if the point, say p, of the final result lies fairly close to the query point q. It would be great if we could discover p very soon in all the relevant lists so that the algorithm can terminate right away. This would become a reality if we could browse the lists synchronously by *distances* as opposed to by ids. In particular, as long as we could access the points of all lists in ascending order of their distances to q (breaking ties by ids), such a p would be easily discovered as its copies in all the lists would definitely emerge *consecutively* in our access order. So all we have to do is to keep counting how many copies of the same point have popped up continuously, and terminate by reporting the point once the count reaches $|W_q|$. At any moment, it is enough to remember only one count, because whenever a new point emerges, it is safe to forget about the previous one.

As an example, assume that we want to perform NN search whose query point q is as shown in Fig. 1, and whose W_q equals $\{c, d\}$. Hence, we will be using the lists of words c and d in Fig. 4. Instead of expanding these lists by ids, the new access order is by distance to q, namely, $p_2, p_3, p_6, p_6, p_5, p_8, p_8$. The processing finishes as soon as the second p_6 comes out, without reading the remaining points. Apparently, if k nearest neighbors are wanted, termination happens after having reported k points in the same fashion.

Distance browsing is easy with R-trees. In fact, the best-first algorithm is exactly designed to output data points in ascending order of their distances to q. However, we must coordinate the execution of best-first on $|W_q|$ R-trees to obtain a global access order. This can be easily achieved by, for example, at each step taking a "peek" at the next point to be returned from each tree, and output the one that should come next globally. This algorithm is expected to work well if the query keyword set W_q is small. For sizable W_q, the large number of random accesses it performs may overwhelm all the gains over the sequential algorithm with merging.

A serious drawback of the R-tree approach is its space cost. Notice that a point needs to be duplicated once for every word in its text description, resulting in very expensive space consumption. In the next section, we will overcome the problem by designing a variant of the inverted index that supports compressed coordinate embedding.

6 Spatial Inverted List

The *spatial inverted list* (SI-index) is essentially a compressed version of an I-index with embedded coordinates as described in Section 5. Query processing with an SI-index can be done either by merging, or together with R-trees in a distance browsing manner. Furthermore, the compression eliminates the defect of a conventional I-index such that an SI-index consumes much less space.

6.1 The Compression Scheme

Compression is already widely used to reduce the size of an inverted index in the conventional context where each inverted list contains only ids. In that case, an effective approach is to record the *gaps* between consecutive ids, as opposed to the precise ids. For example, given a set S of integers $\{2, 3, 6, 8\}$, the gap-keeping approach will store $\{2, 1, 3, 2\}$ instead, where the ith value ($i \geq 2$) is the difference between the ith and $(i-1)$th values in the original S. As the original S can be precisely reconstructed, no information is lost. The only overhead is that decompression incurs extra computation cost, but such cost is negligible compared to the overhead of I/Os. Note that gap-keeping will be much less beneficial if the integers of S are not in a sorted order. This is because the space saving comes from the hope that gaps would be much smaller (than the original values) and hence could be represented with fewer bits. This would not be true had S not been sorted.

Compressing an SI-index is less straightforward. The difference here is that each element of a list, a.k.a. a point p, is a triplet (id_p, x_p, y_p), including both the id and coordinates of p. As gap-keeping requires a sorted order, it can be applied on only one attribute of the triplet. For example, if we decide to sort the list by ids, gap-keeping on ids may lead to good space saving, but its application on the x- and y-coordinates would not have much effect.

To attack this problem, let us first leave out the ids and focus on the coordinates. Even though each point has two coordinates, we can convert them into only one so that gap-keeping can be applied effectively. The tool needed is a space filling curve (SFC) such as Hilbert- or Z-curve. SFC converts a multidimensional point to a 1D value such that if two points are close in the original space, their 1D values also tend to be similar. As dimensionality has been brought to 1, gap-keeping works nicely after sorting the (converted) 1D values.

p_6	p_2	p_8	p_4	p_7	p_1	p_3	p_5
12	15	23	24	41	50	52	59

Fig. 5. Converted values of the points in Fig. 1a based on Z-curve.

For example, based on the Z-curve,[2] the resulting values, called *Z-values*, of the points in Fig. 1a are demonstrated in Fig. 5 in ascending order. With gap-keeping, we will store these 8 points as the sequence $12, 3, 8, 1, 7, 9, 2, 7$. Note that as the Z-values of all points can be accurately restored, the exact coordinates can be restored as well.

Let us put the ids back into consideration. Now that we have successfully dealt with the two coordinates with a 2D SFC, it would be natural to think about using a 3D SFC to cope with ids too. As far as space reduction is concerned, this 3D approach may not a bad solution. The problem is that it will destroy the locality of the points in their original space. Specifically, the converted values would no longer preserve the spatial proximity of the points, because ids in general have nothing to do with coordinates.

If one thinks about the purposes of having an id, it will be clear that it essentially provides a token for us to retrieve (typically, from a hash table) the details of an object, e.g., the text description and/or other attribute values. Furthermore, in answering a query, the ids also provide the base for merging. Therefore, nothing prevents us from using a *pseudo-id* internally. Specifically, let us forget about the "real" ids, and instead, assign to each point a pseudo-id that equals its sequence number in the ordering of Z-values. For example, according to Fig. 5, p_6 gets a pseudo-id 0, p_2 gets a 1, and so on. Obviously, these pseudo-ids can co-exist with the "real" ids, which can still be kept along with objects' details.

The benefit we get from pseudo-ids is that sorting them gives the same ordering as sorting the Z-values of the points. This means that gap-keeping will work *at the same time* on both the pseudo-ids and Z-values. As an example that gives the full picture, consider the inverted list of word d in Fig. 4 that contains p_2, p_3, p_6, p_8, whose Z-values are $15, 52, 12, 23$ respectively, with pseudo-ids being $1, 6, 0, 2$, respectively. Sorting the Z-values automatically also puts the pseudo-ids in ascending order. With gap-keeping, the Z-values are recorded as $12, 3, 8, 29$ and the pseudo-ids as $0, 1, 1, 4$. So we can precisely capture the four points with four pairs: $\{(0, 12), (1, 3), (1, 8), (4, 29)\}$.

Since SFC applies to any dimensionality, it is straightforward to extend our compression scheme to any dimensional space. As a remark, we are aware that the ideas of space filling curves and internal ids have also been mentioned in [8] (but not for the purpose of compression). In what follows, we will analyze the space of the SI-index and discuss how

2. By the Z-curve, a point (x, y) is converted to a 1D value by interleaving the bits of x and y from left to right. For example, in Fig. 1a, the x coordinate of p_4 is $2 = 010$ and its y coordinate is $4 = 100$. Hence, the converted 1D value of p_4 equals $\underline{011}\underline{000}$, where the underlined bits come from 010 and the others from 100.

to build a good R-tree on each inverted list. None of these issues is addressed in [8].

Theoretical analysis. Next we will argue from a theoretical perspective that our compression scheme has a low space complexity. As the handling of each inverted list is the same, it suffices to focus on only one of them, denoted as L. Let us assume that the whole data set has $n \geq 1$ points, and r of them appear in L. To make our analysis general, we also take the dimensionality d into account. Also, recall that each coordinate ranges from 0 to t, where t is a large integer. Naively, each pseudo-id can be represented with $O(\log n)$ bits, and each coordinate with $O(\log t)$ bits. Therefore, without any compression, we can represent the whole L in $O(r(\log n + d \log t))$ bits.

Now we start to discuss the space needed to compress L with our solution. First of all, we give a useful fact on gap-keeping in general:

Lemma 1. *Let $v_1, v_2, \ldots, v_r$ be r non-descending integers in the range from 1 to $\lambda \geq 1$. Gap-keeping requires at most $O(r\log(\lambda/r))$ bits to encode all of them.*

Proof. Denote $u_i = v_i - v_{i-1}$ for $i \in [2, r]$, and $u_1 = v_1$. Note that $\{u_1, u_2, \ldots, u_r\}$ is exactly the set of values gap-keeping stores. Each u_i $(1 \leq i \leq r)$ occupies $O(\log u_i)$ bits. Hence, recording all of $u_1, u_2, \ldots, u_r$ requires at most

$$O(\log u_1 + \log u_2 + \cdots + \log u_r) = O\left(\log\left(\prod_{i=1}^{r} u_i\right)\right) \quad (6)$$

bits. A crucial observation is that

$$1 \leq u_1 + u_2 + \cdots + u_r \leq \lambda$$

as all of $v_1, v_2, \ldots, v_r$ are between 1 and λ. Therefore, $\prod_{i=1}^{r} u_i$ is at most $(\lambda/r)^r$. It thus follows that Equation (6) is bounded by $O(r \log(\lambda/r))$. □

As a corollary, we get:

Lemma 2. *Our compression scheme stores L with $O(r(\log(n/r) + \log(t^d/r)))$ bits.*

Proof. Our compression scheme essentially applies gap-keeping to two sets of integers. The first set includes all the pseudo-ids of the points in L, and the second includes their Z-values. Every pseudo-id ranges from 0 to $n-1$, while each Z-value from 0 to $t^d - 1$. Hence, by Lemma 1, the space needed to store all r pseudo-ids is $O(r(\log(n/r))$, and the space needed to store r Z-values is $O(r(\log(t^d/r))$. □

It turns out that the complexity in the above lemma is already the lowest in the worst case, and no storage scheme is able to do any better, as shown in the following lemma. A similar result also holds for conventional inverted list (without coordinates embedding) as mentioned in [19], chapter 15].

Lemma 3. *Any compression scheme must store L with $\Omega(r(\log(n/r) + \log(t^d/r)))$ bits in the worst case.*

Proof. The lower bound can be established with an information-theoretic approach. First, storing n pseudo-ids must take at least $r \log(n/r)$ bits in the worst case. Remember that each pseudo-id can be any integer from 0 to $n-1$, and thus, there are $\binom{n}{r}$ different ways to choose r different pseudo-ids. Whatever storage scheme must at least be able to distinguish that many ways. It thus follows that at least $\log\binom{n}{r} = \Theta(r \log(n/r))$ bits are necessary in the worst case. Second, similar reasoning also applies to the Z-values. Since each Z-value ranges from 0 to $t^d - 1$, any storage scheme thus needs at least $\log\binom{t^d}{r} = \Theta(r \log(t^d/r))$ bits to encode r Z-values in the worst case. This gives our target result. □

Blocking. The SI-index described up to now applies gap-keeping to capture all points continuously in a row. In decompressing, we must scan an inverted list from its beginning even though the point of our interest lies deep down the list (remember that a point cannot be restored without *all* the gaps preceding it being accumulated). This is not a problem for a query algorithm that performs sequential scan on the list. But in some scenarios (e.g., when we would like to build an R-tree on the list, as in the next section), it is very helpful to restore a point anywhere in the list much faster than reading from the beginning every time.

The above concern motivates the design of the *blocked* SI-index, which differs only in that each list is cut into *blocks* each of which holds $\Theta(B)$ points where B is a parameter to be specified later. For example, given a list of $\{p_1, p_2, p_3, p_4, p_5, p_6\}$, we would store it in two blocks $\{p_1, p_2, p_3\}$ and $\{p_4, p_5, p_6\}$ if the block size is 3. Gap-keeping is now enforced within each block separately. For example, in block $\{p_1, p_2, p_3\}$, we will store the exact pseudo-id and Z-value of p_1, the gaps of p_2 (from p_1) in its pseudo-id and Z-value, respectively, and similarly, the gaps of p_3 from p_2. Apparently, blocking allows to restore all the points in a block locally, as long as the starting address of the block is available. It is no longer necessary to always scan from the beginning.

Since we need to keep the exact values of $\Theta(r/B)$ points, the space cost increases by an addictive factor of $\Theta(\frac{r}{B}(\log n + d \log t))$. This, however, does not affect the overall space complexity in Lemma 2 if we choose B as a polynomial function of r, i.e., $B = r^c$ for any positive $c < 1$. In our experiments, the size of B is roughly $\sqrt{r}$, namely, the value of B can even vary for different inverted lists (i.e., a block may occupy a different number of disk pages). Finally, in a blocked SI-index, each inverted list can also be sequentially accessed from the beginning, as long as we put all its blocks at consecutive pages.

6.2 Building R-Trees

Remember that an SI-index is no more than a compressed version of an ordinary inverted index with coordinates embedded, and hence, can be queried in the same way as described in Section 3.2, i.e., by merging several inverted lists. In the sequel, we will explore the option of indexing each inverted list with an R-tree. As explained in Section 3.2, these trees allow us to process a query by distance browsing, which is efficient when the query keyword set W_q is small.

Our goal is to let each block of an inverted list be directly a leaf node in the R-tree. This is in contrast to the alternative approach of building an R-tree that shares nothing with the inverted list, which wastes space by duplicating each point in the inverted list. Furthermore, our goal is to offer two

search strategies *simultaneously*: merging (Section 3.2) and distance browsing (Section 5).

As before, merging demands that points of all lists should be ordered following the same principle. This is not a problem because our design in the previous section has laid down such a principle: ascending order of Z-values. Moreover, this ordering has a crucial property that conventional id-based ordering lacks: preservation of spatial proximity. The property makes it possible to build good R-trees without destroying the Z-value ordering of any list. Specifically, we can (carefully) group consecutive points of a list into MBRs, and incorporate all MBRs into an R-tree. The proximity-preserving nature of the Z-curve will ensure that the MBRs are reasonably small when the dimensionality is low. For example, assume that an inverted list includes all the points in Fig. 5, sorted in the order shown. To build an R-tree, we may cut the list into 4 blocks $\{p_6, p_2\}$, $\{p_8, p_4\}$, $\{p_7, p_1\}$, and $\{p_3, p_5\}$. Treating each block as a leaf node results in an R-tree identical to the one in Fig. 3a. Linking all blocks from left to right preserves the ascending order of the points' Z-values.

Creating an R-tree from a space filling curve has been considered by Kamel and Faloutsos [16]. Different from their work, we will look at the problem in a more rigorous manner, and attempt to obtain the optimal solution. Formally, the underlying problem is as follows. There is an inverted list L with, say, r points $p_1, p_2, \ldots, p_r$, sorted in ascending order of Z-values. We want to divide L into a number of disjoint blocks such that (i) the number of points in each block is between B and $2B-1$, where B is the block size, and (ii) the points of a block must be consecutive in the original ordering of L. The goal is to make the resulting MBRs of the blocks as small as possible.

How "small" an MBR is can be quantified in a number of ways. For example, we can take its area, perimeter, or a function of both. Our solution, presented below, can be applied to any quantifying metric, but our discussion will use area as an example. The *cost* of a dividing scheme of L is thus defined as the sum of the areas of all MBRs. For notational convenience, given any $1 \leq i \leq j \leq r$, we will use $C[i, j]$ to denote the cost of the optimal division of the subsequence $p_i, p_{i+1}, \ldots, p_j$. The aim of the above problem is thus to find $C[1, r]$. We also denote by $A[i, j]$ the area of the MBR enclosing $p_i, p_{i+1}, \ldots, p_j$.

Now we will discuss the properties of $C[i, j]$. There are $j - i + 1$ points from p_i to p_j. So $C[i, j]$ is undefined if $j - i + 1 < B$, because we will never create a block with less than B points. Furthermore, in the case where $j - i + 1 \in [B, 2B - 1]$, $C[i, j]$ can be immediately solved as the area of the MBR enclosing all the $j - i + 1$ points. Hence, next we will focus on the case $j - i + 1 \geq 2B$.

Notice that when we try to divide the set of points $\{p_i, p_{i+1}, \ldots, p_j\}$, there are at most $B - 1$ ways to decide which points should be in the same block together with the first point p_i. Specifically, a block of size B must include, besides p_i, also p_{i+1}, p_{i+2}, all the way to p_{i+B-1}. If the block size goes to $B + 1$, then the additional point will have to be p_{i+B}; similarly, to get a block size of $B + 2$, we must also put in p_{i+B+1} and so on, until the block size reaches the maximum $2B - 1$. Regardless of the block size, the remaining points (that are not together with p_1) constitute a smaller set on which the division problem needs to be solved recursively. The total number of choices may be less than $B - 1$ because care must be taken to make sure that the number of those remaining points is at least B. In any case, $C[i, j]$ equals the lowest cost of all the permissible choices, or formally:

$$C[i, j] = \min_{k=i+B-1}^{\min\{i+2B-2, j+1-B\}} (A[i, k] + C[k + 1, j]). \quad (7)$$

The equation indicates the existence of solutions based on dynamic programming. One can easily design an algorithm that runs in $O(Br^2)$ time: it suffices to derive $C[i, j]$ in ascending order of the value of $j - i$, namely, starting with those with $j - i = 2B$, followed by those with $j - i = 2B + 1$, and so on until finishing at $j - i = r - 1$. We can significantly improve the computation time to $O(Br)$, by the observation that j can be fixed to r throughout the computation in order to obtain $C[1, r]$ eventually.

We have finished explaining how to build the leaf nodes of an R-tree on an inverted list. As each leaf is a block, all the leaves can be stored in a blocked SI-index as described in Section 6.1. Building the nonleaf levels is trivial, because they are invisible to the merging-based query algorithms, and hence, do not need to preserve any common ordering. We are free to apply any of the existing R-tree construction algorithms. It is noteworthy that the nonleaf levels add only a small amount to the overall space overhead because, in an R-tree, the number of nonleaf nodes is by far lower than that of leaf nodes.

7 Experiments

In the sequel, we will experimentally evaluate the practical efficiency of our solutions to NN search with keywords, and compare them against the existing methods.

Competitors. The proposed SI-index comes with two query algorithms based on merging and distance browsing respectively. We will refer to the former as *SI-m* and the other as *SI-b*. Our evaluation also covers the state-of-the-art *IR^2-tree*; in particular, our IR^2-tree implementation is the fast variant developed in [12], which uses longer signatures for higher levels of tree. Furthermore, we also include the method, named *index file R-tree* (*IFR*) henceforth, which, as discussed in Section 5, indexes each inverted list (with coordinates embedded) using an R-tree, and applies distance browsing for query processing. *IFR* can be regarded as an uncompressed version of *SI-b*.

Data. Our experiments are based on both synthetic and real data. The dimensionality is always 2, with each axis consisting of integers from 0 to 16,383. The synthetic category has two data sets: *Uniform* and *Skew*, which differ in the distribution of data points, and in whether there is a correlation between the spatial distribution and objects' text documents. Specifically, each data set has 1 million points. Their locations are uniformly distributed in *Uniform*, whereas in *Skew*, they follow the Zipf distribution.[3] For both data sets, the vocabulary has 200 words, and each word

3. We create each point independently by generating each of its coordinates (again, independently) according to Zipf.

TABLE 1
Data Set Statistics

	number of points	vocabulary size	average number of objects per word	average number of words per object
Uniform	1 million	200	50k	10
Skew	1 million	200	50k	10
Census	20847	292255	33	461

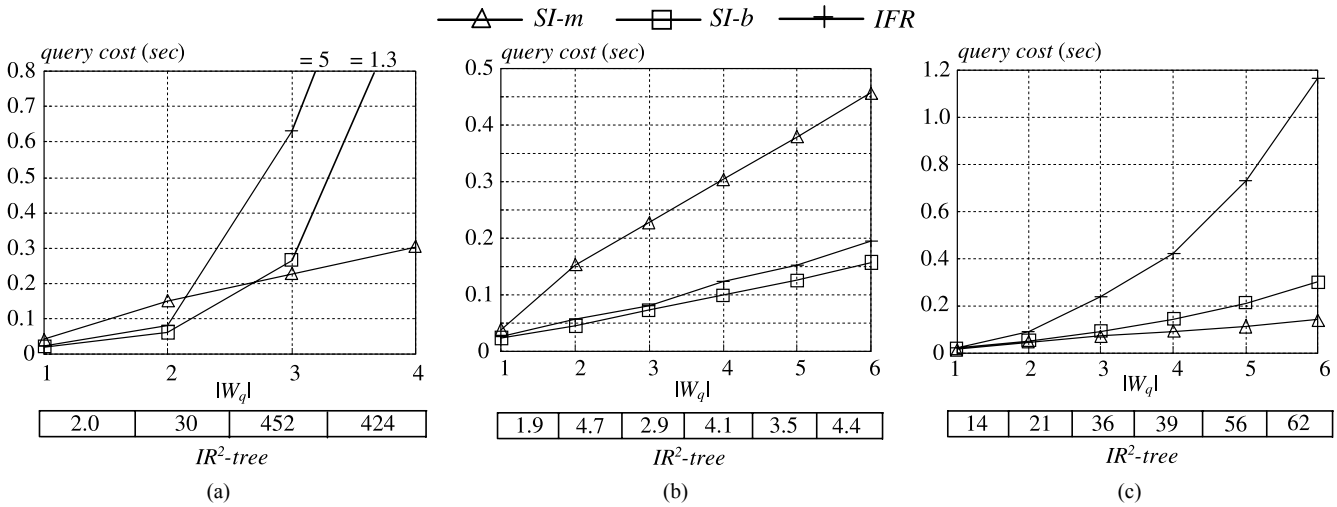


Fig. 6. Query time versus the number of keywords $|W_q|$: (a) Data set *Uniform*, (b) *Skew*, (c) *Census*. The number k of neighbors retrieved is 10.

appears in the text documents of 50k points. The difference is that the association of words with points is completely random in *Uniform*, while in *Skew*, there is a pattern of "word-locality": points that are spatially close have almost identical text documents.

Our real data set, referred to as *Census* below, is a combination of a spatial data set published by the US Census Bureau,[4] and the web pages from *Wikipedia*.[5] The spatial data set contains 20,847 points, each of which represents a county subdivision. We use the name of the subdivision to search for its page at *Wikipedia*, and collect the words there as the text description of the corresponding data point. All the points, as well as their text documents, constitute the data set *Census*. The main statistics of all of our data sets are summarized in Table 1.

Parameters. The page size is always 4,096 bytes. All the SI-indexes have a block size of 200 (see Section 6.1 for the meaning of a block). The parameters of IR^2*-tree* are set in exactly the same way as in [12]. Specifically, the tree on *Uniform* has 3 levels, whose signatures (from leaves to the root) have respectively 48, 768, and 840 bits each. The corresponding lengths for *Skew* are 48, 856, and 864. The tree on *Census* has two levels, whose lengths are 2,000 and 47,608, respectively.

Queries. As in [12], we consider NN search with the AND semantic. There are two query parameters: (i) the number k of neighbors requested, and (ii) the number $|W_q|$ of keywords. Each *workload* has 100 queries that have the same parameters, and are generated independently as follows. First, the query location is uniformly distributed in the data space. Second, the set W_q of keywords is a random subset (with the designated size $|W_q|$) of the text description of a point randomly sampled from the underlying data set. We will measure the query cost as the total I/O time (in our system, on average, every sequential page access takes about 1 milli-second, and a random access is around 10 times slower).

Results on query efficiency. Let us start with the query performance with respect to the number of keywords $|W_q|$. For this purpose, we will fix the parameter k to 10, i.e., each query retrieves 10 neighbors. For each competing method, we will report its average query time in processing a workload. The results are shown in Fig. 6, where (a), (b), (c) are about data sets *Uniform*, *Skew*, and *Census*, respectively. In each case, we present the I/O time of IR^2*-tree* separately in a table, because it is significantly more expensive than the other solutions. The experiment on

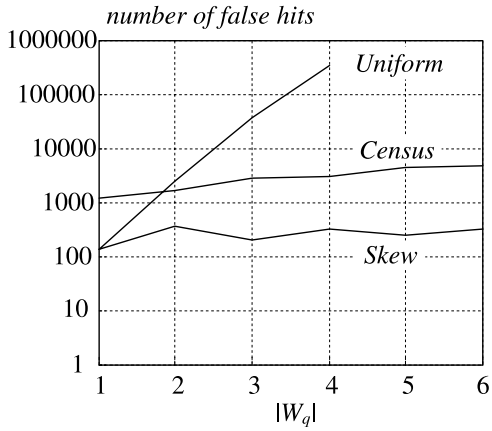


Fig. 7. Number of false hits of IR^2*-tree*.

4. *http://www.census.gov/geo/www/gazetteer/places2k.html*, and follow the link "County Subdivisions".

5. *http://en.wikipedia.org*.

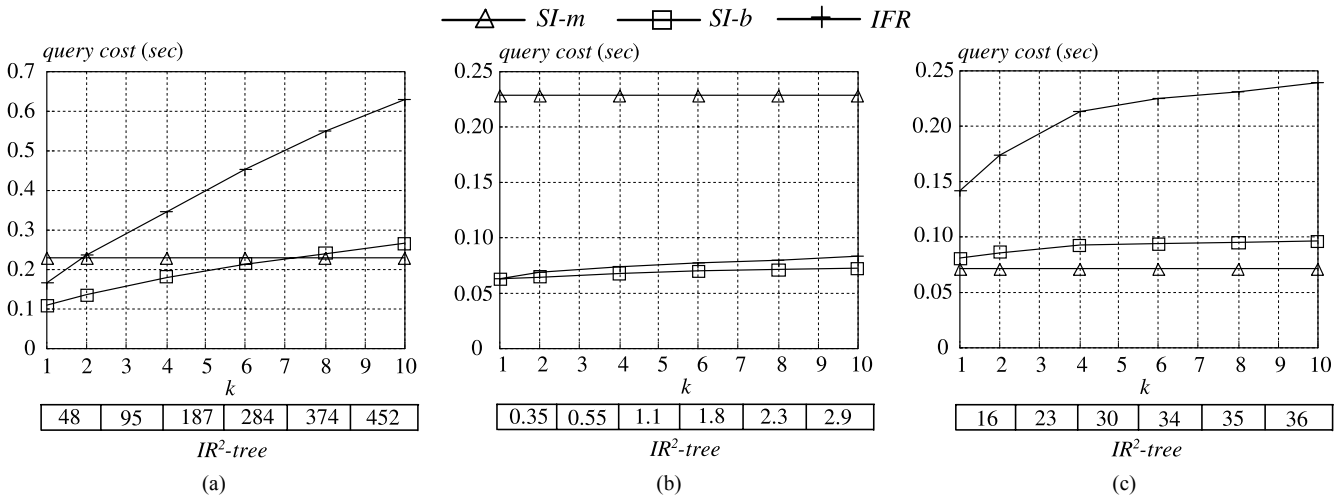


Fig. 8. Query time versus the number of k of neighbors returned: (a) data set *Uniform*, (b) *Skew*, (c) *Census*. The number $|W_q|$ of query keywords is 3.

Uniform inspects $|W_q|$ up to 4, because almost all queries with greater $|W_q|$ return no result at all.

The fastest method is either *SI-m* or *SI-b* in all cases. In particular, *SI-m* is especially efficient on *Census* where each inverted list is relatively small (this is hinted from the column "the number objects per word" in Table 1), and hence, index-based search is not as effective as simple scans. The behavior of the two algorithms on *Uniform* very well confirms the intuition that distance browsing is more suitable when $|W_q|$ is small, but is outperformed by merging when W_q is sizable. On *Skew*, *SI-b* is significantly better than *SI-m* due to the "word-locality" pattern. As for *IFR*, its behavior in general follows that of *SI-b* because they differ only in whether compression is performed. The superiority of *SI-b* stems from its larger node capacity.

IR^2*-tree*, on the other hand, fails to give real time answers, and is often slower than our solutions by a factor of orders of magnitude, particularly on *Uniform* and *Census* where word-locality does not exist. As analyzed in Section 3.1, the deficiency of *IR*^{2}*-tree* is mainly caused by the need to verify a vast number of false hits. To illustrate this, Fig. 7 plots the average false hit number per query (in the experiments of Fig. 6) as a function of $|W_q|$. We see an exponential escalation of the number on *Uniform* and *Census*, which explains the drastic explosion of the query cost on those data sets. Interesting is that the number of false hits fluctuates[6] a little on *Skew*, which explains the fluctuation in the cost of *IR*^{2}*-tree* in Fig. 6b.

Next, we move on to study the other query parameter k (the number of neighbors returned). The experiments for this purpose are based on queries with $|W_q| = 3$. As before, the average query time of each method in handling a workload is reported. Figs. 8a, 8b, and 8c give the results on *Uniform*, *Skew*, and *Census*, respectively. Once again, the costs of *IR*^{2}*-tree* are separated into tables. In these experiments, the best approach is still either *SI-m* or *SI-b*. As expected, the cost of *SI-m* is not affected by k, while those of the other solutions all increase monotonically. The relative superiority of alternative methods, in general, is similar to that exhibited in Fig. 6. Perhaps worth pointing out is that, for all distributions, distance browsing appears to be the most efficient approach when k is small.

Results on space consumption. We will complete our experiments by reporting the space cost of each method on each data set. While four methods are examined in the experiments on query time, there are only three as far as space is concerned. Remember that *SI-m* and *SI-b* actually deploy the same SI-index and hence, have the same space cost. In the following, we will refer to them collectively as *SI-index*.

Fig. 9 gives the space consumption of *IR*^{2}*-tree*, *SI-index*, and *IFR* on data sets *Uniform*, *Skew*, and *Census*, respectively. As expected, *IFR* incurs prohibitively large space cost, because it needs to duplicate the coordinates of a data point p as many times as the number of distinct words in the text description of p. As for the other methods, *IR*^{2}*-tree* appears to be slightly more space efficient, although such an advantage does not justify its expensive query time, as shown in the earlier experiments.

Summary. The SI-index, accompanied by the proposed query algorithms, has presented itself as an excellent trade-off between space and query efficiency. Compared to *IFR*, it consumes significantly less space, and yet, answers queries

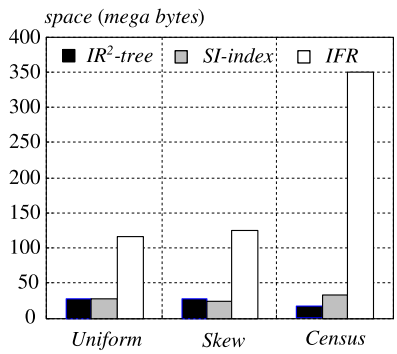


Fig. 9. Comparison of space consumption.

6. Such fluctuation is not a surprise because, as discussed in Section 3.1, the number of false hits is determined by two factors that may cancel each other: (i) how many data points are closer than the kth NN reported, and (ii) the false hit probability. While the former factor increases with the growth of $|W_q|$, the latter actually decreases.

much faster. Compared to IR^2-*tree*, its superiority is overwhelming since its query time is typically lower by a factor of orders of magnitude.

8 Conclusions

We have seen plenty of applications calling for a search engine that is able to efficiently support novel forms of spatial queries that are integrated with keyword search. The existing solutions to such queries either incur prohibitive space consumption or are unable to give real time answers. In this paper, we have remedied the situation by developing an access method called *the spatial inverted index* (SI-index). Not only that the SI-index is fairly space economical, but also it has the ability to perform keyword-augmented nearest neighbor search in time that is at the order of dozens of milli-seconds. Furthermore, as the SI-index is based on the conventional technology of inverted index, it is readily incorporable in a commercial search engine that applies massive parallelism, implying its immediate industrial merits.

Acknowledgments

This work was supported in part by (i) projects GRF 4166/10, 4165/11, and 4164/12 from HKRGC, and (ii) the WCU (World Class University) program under the National Research Foundation of Korea, and funded by the Ministry of Education, Science and Technology of Korea (Project No: R31-30007).

References

[1] S. Agrawal, S. Chaudhuri, and G. Das, "Dbxplorer: A System for Keyword-Based Search over Relational Databases," *Proc. Int'l Conf. Data Eng. (ICDE)*, pp. 5-16, 2002.
[2] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger, "The R*-tree: An Efficient and Robust Access Method for Points and Rectangles," *Proc. ACM SIGMOD Int'l Conf. Management of Data*, pp. 322-331, 1990.
[3] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan, "Keyword Searching and Browsing in Databases Using Banks," *Proc. Int'l Conf. Data Eng. (ICDE)*, pp. 431-440, 2002.
[4] X. Cao, L. Chen, G. Cong, C.S. Jensen, Q. Qu, A. Skovsgaard, D. Wu, and M.L. Yiu, "Spatial Keyword Querying," *Proc. 31st Int'l Conf. Conceptual Modeling (ER)*, pp. 16-29, 2012.
[5] X. Cao, G. Cong, and C.S. Jensen, "Retrieving Top-k Prestige-Based Relevant Spatial Web Objects," *Proc. VLDB Endowment*, vol. 3, no. 1, pp. 373-384, 2010.
[6] X. Cao, G. Cong, C.S. Jensen, and B.C. Ooi, "Collective Spatial Keyword Querying," *Proc. ACM SIGMOD Int'l Conf. Management of Data*, pp. 373-384, 2011.
[7] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal, "The Bloomier Filter: An Efficient Data Structure for Static Support Lookup Tables," *Proc. Ann. ACM-SIAM Symp. Discrete Algorithms (SODA)*, pp. 30-39, 2004.
[8] Y.-Y. Chen, T. Suel, and A. Markowetz, "Efficient Query Processing in Geographic Web Search Engines," *Proc. ACM SIGMOD Int'l Conf. Management of Data*, pp. 277-288, 2006.
[9] E. Chu, A. Baid, X. Chai, A. Doan, and J. Naughton, "Combining Keyword Search and Forms for Ad Hoc Querying of Databases," *Proc. ACM SIGMOD Int'l Conf. Management of Data*, 2009.
[10] G. Cong, C.S. Jensen, and D. Wu, "Efficient Retrieval of the Top-k Most Relevant Spatial Web Objects," *PVLDB*, vol. 2, no. 1, pp. 337-348, 2009.
[11] C. Faloutsos and S. Christodoulakis, "Signature Files: An Access Method for Documents and Its Analytical Performance Evaluation," *ACM Trans. Information Systems*, vol. 2, no. 4, pp. 267-288, 1984.
[12] I.D. Felipe, V. Hristidis, and N. Rishe, "Keyword Search on Spatial Databases," *Proc. Int'l Conf. Data Eng. (ICDE)*, pp. 656-665, 2008.
[13] R. Hariharan, B. Hore, C. Li, and S. Mehrotra, "Processing Spatial-Keyword (SK) Queries in Geographic Information Retrieval (GIR) Systems," *Proc. Scientific and Statistical Database Management (SSDBM)*, 2007.
[14] G.R. Hjaltason and H. Samet, "Distance Browsing in Spatial Databases," *ACM Trans. Database Systems*, vol. 24, no. 2, pp. 265-318, 1999.
[15] V. Hristidis and Y. Papakonstantinou, "Discover: Keyword Search in Relational Databases," *Proc. Very Large Data Bases (VLDB)*, pp. 670-681, 2002.
[16] I. Kamel and C. Faloutsos, "Hilbert R-Tree: An Improved R-Tree Using Fractals," *Proc. Very Large Data Bases (VLDB)*, pp. 500-509, 1994.
[17] J. Lu, Y. Lu, and G. Cong, "Reverse Spatial and Textual k Nearest Neighbor Search," *Proc. ACM SIGMOD Int'l Conf. Management of Data*, pp. 349-360, 2011.
[18] S. Stiassny, "Mathematical Analysis of Various Superimposed Coding Methods," *Am. Doc.*, vol. 11, no. 2, pp. 155-169, 1960.
[19] J.S. Vitter, "Algorithms and Data Structures for External Memory," *Foundation and Trends in Theoretical Computer Science*, vol. 2, no. 4, pp. 305-474, 2006.
[20] D. Zhang, Y.M. Chee, A. Mondal, A.K.H. Tung, and M. Kitsuregawa, "Keyword Search in Spatial Databases: Towards Searching by Document," *Proc. Int'l Conf. Data Eng. (ICDE)*, pp. 688-699, 2009.
[21] Y. Zhou, X. Xie, C. Wang, Y. Gong, and W.-Y. Ma, "Hybrid Index Structures for Location-Based Web Search," *Proc. Conf. Information and Knowledge Management (CIKM)*, pp. 155-162, 2005.

Yufei Tao is currently a full professor at the Chinese University of Hong Kong. He also holds a visiting professor position, under the World Class University (WCU) program of the Korean government, at the Korea Advanced Institute of Science and Technology (KAIST). He is an associate editor of the *ACM Transactions on Database Systems* (TODS), and of the *IEEE Transactions on Knowledge and Data Engineering* (TKDE). He is/was a PC co-chair of ICDE 2014, a PC co-chair of SSTD 2011, an area PC chair of ICDE 2011, and a senior PC member of CIKM 2010, 2011, and 2012.

Cheng Sheng received the PhD degree in computer science from the Chinese University of Hong Kong in 2012 and the BSc degree in computer science from Fudan University in 2008. He is currently a software engineer at Google Switzerland. His research focuses on algorithms in database systems with nontrivial theoretical guarantees.