Top-k Point of Interest Retrieval Using Standard Indexes

Anders Skovsgaard
Department of Computer Science
Aarhus University, Denmark
anderssk@cs.au.dk

Christian S. Jensen
Department of Computer Science
Aalborg University, Denmark
csj@cs.aau.dk

ABSTRACT

With the proliferation of Internet-connected, location-aware mobile devices, such as smartphones, we are also witnessing a proliferation and increased use of map-based services that serve information about relevant Points of Interest (PoIs) to their users.

We provide an efficient and practical foundation for the processing of queries that take a keyword and a spatial region as arguments and return the k most relevant PoIs that belong to the region, which may be the part of the map covered by the user's screen. The paper proposes a novel technique that encodes the spatio-textual part of a PoI as a compact bit string. This technique extends an existing spatial encoding to also encode the textual aspect of a PoI in compressed form. The resulting bit strings may then be indexed using index structures such as B-trees or hashing that are standard in DBMSs and key-value stores. As a result, it is straightforward to support the proposed functionality using existing data management systems. The paper also proposes a novel top-k query algorithm that merges partial results while providing an exact result.

An empirical study with real-world data indicates that the proposed techniques enable excellent indexing and query execution performance on a standard DBMS.

Categories and Subject Descriptors

H.2.2 [**Database Management**]: Physical Design—*access methods*; H.2.8 [**Database Management**]: Database Applications—*spatial databases and GIS*

General Terms

Algorithms

Keywords

Spatio-textual index, spatial keyword query, top-k query, S2 cell, textual encoding, key-value store

1. INTRODUCTION

The proliferation of Internet-enabled, geo-positioned mobile devices has created an increased demand for local information. Thus, we are also witnessing an increased proliferation and use of different map-based services that serve local information. Local information is typically organized around Points of Interest (PoIs) that may be of different types, e.g., hotels, restaurants, and parks, that include exact locations, names, and textual descriptions.

Map-based services allow users to retrieve such different types of PoIs near them. For example, users may look for hotels or restaurants in neighborhoods where they are currently present or are going to be at a later time. Typical map-based functionality supports the retrieval of relevant PoIs that belong to the part of space visible on the user's screen. This functionality supports users who wish to browse or explore their surroundings and is different from the retrieval of PoIs that are closest to an exact user location.

Several proposals exist that are capable of finding the most relevant places that are closest to an exact location. They are typically either R-tree based [3, 7, 8, 13, 17, 18, 20–22], grid based [15, 19], or space filling curve based [5, 6]. The proposals either support Boolean top-k or queries, or they support top-k queries with a ranking function. However, they all suffer from two limitations: (i) they involve the use of a special index structure, making it difficult to leverage existing data management systems, and (ii) they focus on finding the top-k most relevant objects, that are closest to an exact location instead of providing the top-k most relevant objects in a region.

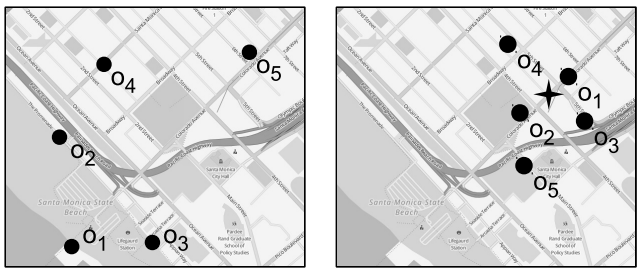


Figure 1: Top-5 results for a region query (left) and a nearest neighbor query (right)

We study a query that returns the top-k most relevant PoIs in a spatial region. An example is given in Figure 1, that shows the top-5 most relevant objects for a region and the result of a nearest-neighbor type query. The left example supports exploratory user behavior in the region, since the top-5 most relevant objects may be located anywhere in the region. In the example, the three best places are near the beach. This approach is used by all of the major providers of map-based services, including Google Maps, Bing Maps, and Yahoo! Maps. The example to the right retrieves objects close to the query location and is fundamentally different.

The paper makes four contributions.

SIGSPATIAL'14, November 04 - 07 2014, Dallas/Fort Worth, TX, USA

http://dx.doi.org/10.1145/2666310.2666399.

- We propose and give a precise definition of a query that finds the top-k most relevant objects in a spatial region with respect to their textual descriptions. The query is similar to the approach used by major providers of map-based services. A baseline algorithm is given that modifies state-of-the-art algorithms for nearest neighbor queries to support the proposed region query.
- We propose a novel approach to the representation and indexing of spatio-textual objects using standard indexes that are widely available in data management systems such as traditional DBMSs and key-value stores. The locations and textual descriptions of the objects are first encoded into bit strings with a spatial and a textual part. The spatial part is encoded using an existing encoding scheme, while the encoding of the textual part employs a hash function. Collisions among the bit strings that encode the text may occur, and a parameter is introduced that controls the balance between query execution time and space consumption. The encoding scheme makes it possible to leverage standard indexes such as B-trees for the indexing of the spatio-textual data.
- We present a query processing algorithm that computes exact results for the proposed query. The algorithm queries the spatio-textual objects using the indexed bit strings. Query regions may overlap several grid cells, which are then merged while guaranteeing exact query results.
- We report on an experimental study of the proposed techniques using real-world data. The study suggests that the proposed solution is capable of efficiently supporting the proposed query.

The remainder of the paper is structured as follows. Section 2 covers related work. The problem definition is given in Section 3. Section 4 presents the proposed solution, encompassing the spatio-textual bit string encoding and indexing strategy followed by an exact query processing algorithm. The experimental evaluation is given in Section 5. Finally, we conclude and offer research directions in Section 6.

2. RELATED WORK

Recently, substantial research efforts on geo-textual indexes have been reported. The proposed solutions often use a combination of the R-tree [12] or a variant [2] for the spatial indexing and inverted files for indexing the text [7, 13, 17, 18, 20, 22]. Others use bitmap files for text indexing [3, 8, 21]. Solutions using grids [15, 19] and space-filling-curves [5, 6] combined with inverted files have also been proposed. The spatio-textual parts are combined in three different ways: (i) either closely combined, with no clear separation [6–8, 13, 15, 17, 20, 21], (ii) with the spatial index followed by the text index [3, 5, 19, 22], (iii) or with the text index followed by the spatial index [18, 19, 22]. A comprehensive experimental evaluation of geo-textual indexes is also available [4].

Most of the related work targets Boolean queries and does not return ranked results. This paper focuses on providing a ranked result to provide a size-limited result. With the vast amounts of data that may exist in a region, it is often not helpful to return all objects in a region that contain a given term.

Some of the related work is able to efficiently return the top-k most relevant nearest neigbors [7, 17, 18, 20]. These works all use the R-tree and the query processing algorithms may be modified to find the top-k most relevant objects in a region. A baseline algorithm that uses a geo-textual index is given in Section 3.2. However, this related work uses custom index structures, while this paper proposes a technique that may be employed with existing standard indexing techniques. Despite the substantial research on answering spatio-textual queries, to the best of our knowledge, no existing work addresses encoding of spatio-textual objects in combination with existing indexing techniques.

Multi-dimensional spaces can be mapped to a one-dimensional space in order to support the indexing of multi-dimensional points in a standard DBMS. The mapping of multi-dimensional space into a one-dimensional space is often done using a space-filling curve such as a Z-curve or a Hilbert-curve [23]. The existing techniques consider only the spatial dimension, whereas we propose a new problem that considers both the spatial and the textual properties of the objects.

The Google S2 Geometry Library [11] provides an open-source implementation that maps two-dimensional objects on the Earth's surface to a one-dimensional representation. It encloses the Earth sphere in a cube and imposes a quad-tree [10] type partitioning to each of the 6 faces. The cells are encoded and decoded by means of a Hilbert curve [14]. This provides a bit string representation of a location; this paper leverages and extends this work.

Hash functions can map text strings of arbitrary lengths to fixed-length hash values [16]. When two different strings produce the same hash value, a collision occurs. As the length of the hash value decreases, the number of collisions increases. A perfect hash functions map the input to unique hash values and is thus collision-free. Hash functions are used in the proposed solution to limit the number of textual descriptions and their lengths.

3. PRELIMINARIES

We proceed with giving a formal definition of the problem addressed in the paper followed by a baseline algorithm that modifies state-of-the-art algorithms.

3.1 Problem Definition

Let D be a set of spatio-textual objects $o = (\lambda, doc)$, where λ is a point location, and doc is a text document. Let D_R be the set of objects in D that are inside the spatial region R,

A top-k spatio-textual region (TkSTR) query TkSTR = $(R, keyword, k)$ returns an ordered list of k objects from D_R that are the most relevant to $keyword$ according to a textual relevance function computed by language models similar to the one presented in related work [7]. For ease of understanding, we use the term frequency function $tf(t, o.doc)$, that returns the number of occurrences of term t in the text document $o.doc$. However, other textual relevance functions may be applied.

Definition 1. With the above definitions, we can define the result of the TkSTR query with arguments $(R, keyword, k)$ as follows. The query returns k objects from D_R, $o_1, \ldots, o_k$, where

$$tf(keyword, o_i.doc) \geq tf(keyword, o_{i+1}.doc), i = 1, \ldots, k-1.$$

Further, there does not exist an object $o \in D_R$ such that

$$tf(keyword, o.doc) > tf(keyword, o_i.doc), i = 1, \ldots, k.$$

Example 1. Figure 2 describes five objects $o_1, \ldots, o_5$, and Table 2 shows the document-term matrix of their documents.

Given the query Q with $Q.R$ be the spatial region R in Figure 2, $Q.keyword = pizza$, and $Q.k = 3$ the ordered result is (o_4, o_2, o_5). No object inside R has a higher frequency than the

objects in the ordered result set. The object o_3 has the highest term frequency but lies outside $Q.R$.

For the same arguments but with $Q.keyword = sushi$ the ordered result is (o_1, o_5, o_2) □

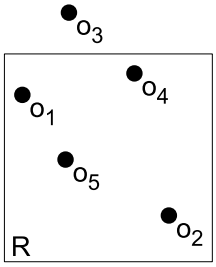


Figure 2: A Spatial Region and Five Objects

	pizza	*sushi*	*shoe*
$o_1.doc$	2	5	1
$o_2.doc$	4	3	4
$o_3.doc$	7	2	0
$o_4.doc$	5	2	2
$o_5.doc$	3	4	6

Table 2: Term Frequencies of the Objects in Figure 2

3.2 Baseline Algorithm

No algorithm has yet been proposed for the efficient computation of the TkSTR query. Thus, we consider how to modify existing techniques to process the TkSTR query before we present the proposed solution. Recent related work proposes geo-textual index structures that aim to enable efficient computation of the top-k most relevant nearest neighbors [7, 17, 18, 20]. They all employ R-tree based structures, which may also be used to return the result for a region query.

Each of the related works has a parameter α that allows balancing spatial proximity and textual relevancy. The result of the TkSTR query does not depend on the spatial proximity of the objects. Therefore, α may be set to 0 in some studies [7,18,20] and to 1 in another study [17] in order to eliminate the spatial proximity constraint.

The query processing algorithms from related work take a point location as argument. In contrast, the TkSTR query employs a query region. To support a query region, we use a point location placed at the center of the TkSTR query region, and we only consider objects in this region. Thus, the search is only expanded from the point location within this region when iterating through nearest neighbors. The query processing terminates when a nearest neighbor is encountered that is further away from the query point than the longest distance from the query point to the furthest away border of the query region. Then, all objects inside the query region have been considered, and no additional relevant objects exist.

The query processing algorithms in related work have to consider each nearest neighbor until the $Q.k$ most relevant objects have been found. In the worst case, all objects inside the query region have to be examined since the most relevant object may be located anywhere in the query region. However, the nodes of the R-trees in the existing proposals are augmented with information about the best term frequencies in their subtrees. This information may be used to prune some of the R-tree nodes. Since, it is not known which and how many objects contain the specific term frequencies, expensive full node scans result.

The difficulty in providing an efficient baseline using existing techniques combined with the requirement of a custom index structure being available suggests that it is relevant to consider the invention of a new solution.

4. PROPOSED SOLUTION

We present a framework that encodes spatio-textual objects into compact bit strings that may be stored in standard data management systems and indexed using standard index structures that are generally available in such systems. The framework encompasses an exact query processing algorithm capable of computing TkSTR queries by using the bit strings and by combining partial results.

4.1 Spatio-Textual Bit String Encoding

We proceed to describe the encoding of the spatio-textual objects and then provide an overview of the framework.

Modern data management systems generally have a number of indexes that are optimized for processing queries efficiently. Standard indexes include B-trees [1] and hash tables! [16], and some systems also support R-trees [12]. However, many queries are not able to use these standard index structures directly, including the TkSTR query and the top-k most relevant nearest neighbor query.

We aim to develop an approach that can process the TkSTR query by utilizing standard indexes that are available in virtually any data management system such as a DBMS or a key-value store: B-trees or the hash tables. To achieve this, we first encode spatio-textual objects into bit strings that they may be readily indexed by standard indexes. A bit string has two parts: (i) a spatial part that encodes the location of the object being encoded and is used to determine whether the object is located in the query region, and (ii) a textual part that represents the text document of the object and helps determine whether the object indeed contains the query term.

4.1.1 Spatial Encoding

We have the following requirements to the spatial part of the bit string:

1. Compact representation.
2. Fast lookup of regions of arbitrary size and location.
3. Sufficiently high resolution.
4. Uniform query performance.

It is desirable to have a compact bit string to reduce the storage requirements and access cost. We aim at efficiently processing queries independently of the size and location of the query region. Also, we aim at providing a result efficiently for even very small query regions. Finally, the performance should not vary greatly depending on the size or location of the query region.

The Google S2 Geometry Library [11] utilizes a quad-tree style partitioning that yields cells of uniform area at multiple granularities. This satisfies the requirements of supporting arbitrary region sizes and locations. A bit string is created by enclosing the Earth by a cube and by imposing a quad-tree style partitioning on each of the 6 faces of the cube, as illustrated in the example in Figure 3. The quad-tree type partitioning has multiple levels, and the cells are enumerated using a Hilbert curve [14] that makes it possible to efficiently find any cell at any level.

The faces of the cube can be represented with 3 bits, and the 4 cells at each level in the partitioning can be encoded using 2 bits.

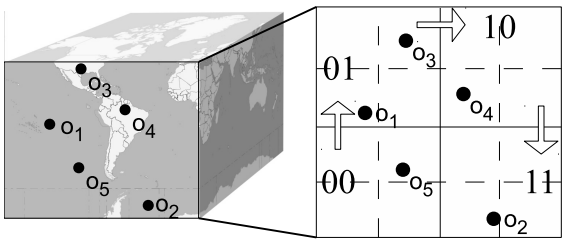


Figure 3: Earth Enclosed in a Cube (left) and a Grid Index for Each of the 6 Faces (right)

With a total surface area of the Earth of 510,072,000 km^2, each of the faces covers an area of 85,012,000 km^2. To provide an accuracy of less than 1 cm^2, 30 levels are required, which can be captured by using 63-bits. We may not need this level of accuracy, and we may reduce the number of levels, thus making the bit string more compact while still providing sufficient resolution. The experimental evaluation studies the impact of different numbers of levels. This fulfils our requirement of achieving a compact bit string.

Example 2. Consider the object o_1 in Figure 3. It is on the first face of the cube which is represented by 3 bits: 000. By following the Hilbert curve in the grid tree, we find that object o_1 is in the cell with position 1 at the first level, which is represented by the bits 01. At the second level, it is in the cell with position 4, which us represented by 00. The complete encoding of the level 2 bit string of o_1 is then 0000100. □

With this approach, we are able to encode the spatial locations of objects as compact bit strings. The resulting bit strings can be indexed using standard indexes. We proceed to extend this approach with a textual bit string representation.

4.1.2 Textual Encoding

Large amounts of PoIs that cover all countries in the world are maintained by providers of map-based services. Thus, the textual descriptions may be in any language, resulting in a large vocabulary. We have two requirements to the textual part of the bit string:

1. Compact representation.
2. Limited number of bit strings.

Having a compact bit string representation instead of string with characters in any language reduces the storage requirements. Since standard indexing techniques like B-trees and hash tables are affected by the number of indexed objects, it is important to have a known and limited number of bit strings. Therefore, we aim at providing a limited number of compact bit strings that may be used for direct lookup in a standard index.

The TkSTR query takes the argument $keyword$, which is a term, e.g., "pizza" or "sushi." By encoding each term in the object document, $o.doc$, we may use the bit strings to produce a result for the TkSTR queries. To encode a term into a bit string of a fixed length, we propose to use the hash value of each term in the textual description. By using a one-way hash function, we achieve both of the above requirements. First, the length of the bit string may be set to be sufficiently short by the choice of the hash function. Second, a hash function produces a fixed number of hash values. Next, it is possible to use any hash function and to use truncation of the hash values to obtain a limited number of compact bit strings. In the experimental evaluation, we study the use of different lengths of the hash values.

Hash functions may introduce collisions, which occur when two or more input text strings produce the same output hash value. An example is shown in Figure 4 where the terms "pizza" and "sushi" both hash to the same value "00".

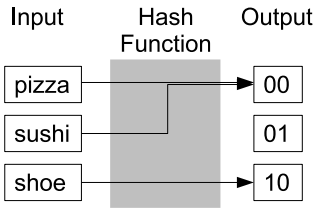


Figure 4: Two Input Strings Produce the Same Output Hash Value

When the hash function produces a small number of output hash values or when the hash values are truncated, it is more likely to see more collisions. However, by increasing the amount and length of the output hash values, the storage requirements and the time to perform indexing and lookups also increase.

By encoding the terms as compact bit strings, we may concatenate these with the spatial bit strings to achieve a single, compact bit string. The resulting compact bit strings hold information about the locations and the textual descriptions of the objects they represent.

Example 3. Consider the object o_1 in Figure 3 and the term frequencies in Table 1. With the hash function in Figure 4, the combined bit string with level 1 spatial resolution for the terms "pizza" and "sushi" is 0000100, while "shoe" encodes to the string 0000110. □

4.1.3 Handling Collisions

The text documents of objects that are located in the same spatial grid cell may contain the same terms, resulting in identical bit strings. We call this a bit string collision. Also, objects located in the same cell, but with different terms, may be encoded to the same bit string since we use one-way hash functions that produce collisions. We call this a term collision. To be able to provide exact result for the queries, we need to be able to distinguish between co-located objects that share the same terms. We proceed to present techniques for handling objects with the two types of collisions: bit string collisions and term collisions.

Bit String Collision Multiple objects may exists in the same grid cell, resulting in the same bit string. This is more likely at the first levels of the grid since they cover larger areas. With the large amounts of data, any grid cell may contain numerous objects. These objects are also contained in grid cells at the lower levels. We want to avoid the duplicate storage of these large amounts of objects.

According to Definition 1, the TkSTR query returns $Q.k$ objects with the highest term frequency. Therefore, in order to provide an efficient and exact result for any given grid cell, it is only necessary to store the $Q.k$ objects with highest term frequency. The remaining objects will not be used to answer queries with a region that exactly matches the size of a grid cell. We propose to set a maximum value, $targetedK \geq Q.k$, for the number of objects to store in each cell. Thus, we can reduce the storage requirements and process all queries with $Q.k \leq targetedK$ without using any cells from the lower levels. Naturally, the value of $targetedK$ should be set sufficiently high in order to provide a useful number of objects in the result set.

The query region may not have the same size of the grid cell and thereby contain the $Q.k$ objects. The query region may be smaller

than the grid cells and even smaller than the grid cells at the lowest level. To support queries with such regions, we store all objects at the lowest level. Thus, the lowest-level grid cells do not only store $targetedK$ objects, but store all objects that are located in the regions of the cells. Since we store all objects at the lowest level, no information is lost, and an exact result may be provided for any query region.

Term Collision By using hashing, we get a limited number of compact bit strings that may reduce storage requirements and the amount of identifiers to index. However, this may introduce collisions as shown in the example in Figure 4. We propose a method that handles the term collisions such that it may be employed using any key-value store.

The grid cells may describe up to $targetedK$ objects for any given term. For the cells at the lowest level, the number may exceed $targetedK$. The mapping of a term to the actual objects can easily be done when there are no term collisions. The hash value simply serves as a unique identifier for the specific term, and it may point directly to a bucket with the objects. Consider Figure 5 where the same input and output as in Figure 4 are used and $targetedK$ is set to 2. The input term "shoe" encodes to "10" and points directly to objects o_5 and o_2 from Figure 3 ordered by the term frequencies from Table 1.

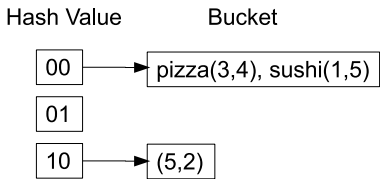


Figure 5: Buckets with Collisions

However, when term collisions occur, we propose to store a separate chain of terms in a bucket since the hash value no longer uniquely identifies the terms. Each of these stored terms identifies the corresponding objects. In Figure 5 both of the terms "pizza" and "sushi" encode to "00." Therefore, the bucket contains both terms along with the corresponding objects. With this approach, we can distinguish between objects that are located in the same grid cell and have the same terms.

4.1.4 Framework Overview

We proceed to give an overview of the framework. First, the data is preprocessed such that each term of the object documents is handled with the proposed spatial and textual encoding techniques as illustrated in Figure 6. The result is a bit string for each term of the object documents. The bit string is then stored and indexed in a standard data management system using a standard indexing technique. A detailed description of the storing of objects is provided in the following section.

The query processing is shown in the lower part of Figure 6 and is covered in detail in Section 4.3. First, the query region and keyword are preprocessed such that the spatial and textual parts of the query are encoded into a bit string. This bit string is then used to perform a lookup in data management system used, which yields a number of objects. Lower levels of the grid structure may have to be searched in order to produce an exact result. Therefore, the bit string is refined and used to perform additional lookups. When enough objects necessary to answer the query have been retrieved, the result is returned.

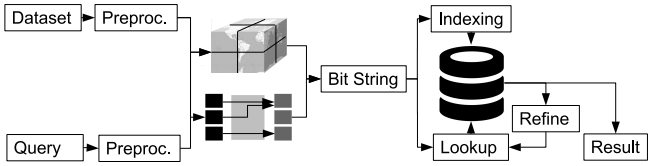


Figure 6: Framework Overview with the Spatial and Textual Encoding (two left figures) and a Standard Data Management System (right figure)

4.2 Storing Objects in a Key-Value Store

Existing DBMS and cloud database technologies all support key-value stores with standard indexing techniques such as B-trees and hash tables. We propose a method that works with all key-value store indexing techniques. The proposed method encodes an object to a bit string that may be represented by data types such as integers or strings, depending on the types available in the system used. When storing the objects, each term of the object documents is encoded into a bit string. To apply the proposed techniques to a key-value store, we store the bit strings as the key. By having an index on the key, we are able to perform lookups of the bit strings efficiently.

Since the query region may be of arbitrary size, we store a bit string for each level of the grid, as seen from the *key* column in Table 3 that describes all objects from Figure 3. If we only stored objects at the finest levels, a large number of cells would have to be fetched to provide a result for queries with large regions.

Without term collisions, any key will at most map to $targetedK$ object references. A standard page of size either 4 or 8 KB can hold 32 bit integer references to 1,000 or 2,000 objects, respectively. To avoid fetching each object when performing query processing, we propose to store the term frequency in a 16 bit integer along with a 32 bit integer for both the latitude and longitude for each object. Thus, the object representation will have the following structure:

$$id(INT32)tf(INT16)lat(INT32)lng(INT32).$$

In total, 14 bytes are required to store this object reference. A page of size 4 or 8 KB can hold approximately 300 or 600 objects, respectively. We assume that $Q.k$ will be much smaller than this number, and we thus also assume that $targetedK$ will be similarly small.

With these storage requirements, we propose to store all object references directly as the values, as illustrated in Table 3. The table describes the objects from Figure 3 ordered by the term frequencies from Table 1. To simplify, we only give the identifier of the object and not the full object representation.

key	$value$
00000	pizza(3,4), sushi(1,5)
00010	shoe(5,2)
0000000	pizza(5), sushi(5)
0000010	shoe(5)
0000100	pizza(3,1), sushi(1,3)
0000110	shoe(1)
0001000	pizza(4), sushi(4)
0001010	shoe(4)
0001100	pizza(2), sushi(2)
0001110	shoe(2)

Table 3: Key-Value Database with $targetedK$ set to 2

When objects produce term collisions, we lose the direct mapping from bit string to a term. The term of a stored object is

required in order to detect when a term collision occurs. Therefore, we also store the terms along with the object representations to avoid fetching the complete textual descriptions of the objects. Term collisions may eventually occur, which is why we store the terms pro-actively for all values. The storing of terms requires more space depending on the number of term collisions. The experimental evaluation provides insight into the storage requirements.

All objects are stored in sorted order with regard to the term frequencies. The insertion algorithm is given in Algorithm 1, which takes a dataset of spatio-textual objects and a grid level as arguments. With no bit string collision, an object can be inserted directly, as shown in Line 10 since no other object with the term exists. When a term is creating a term collision for the first time, the same applies. If the cell contains less than $targetedK$ objects for a given term, the object is inserted, since there is room for it; see Line 13. All objects are inserted in cells at the lowest level.

Algorithm 1: `Insert`(Dataset D, GridLevel gl, Boolean $lowest$)

```
   /* Global variables: */
1  db ← Key-Value Store ;
2  targetedK ← max obj. ref. stored in DB ;        // conf. param.
   /* Local variables: */
3  bs ← empty bit string ;
4  foreach o ∈ D do
5      foreach t ∈ o.doc do
6          bs ← spatialEncode(gl, o.λ) ;
7          bs ← bs ∪ termEncode(t) ;
8          value ← db.lookup(bs) ;
9          if value is empty ∨ value does not contain t then
10             db.insert(o) ;              // no objects with t exists
11         else if value contains t then
12             if lowest ∨ |value| < targetedK then
13                 db.insert(o) ;                    // there is room
14             else if tf(t,o.doc) > value.mintf then
15                 db.sortAndUpdate(o) ;             // more relevant
```

Because we store the term frequency as part of the object representation, dynamic updates are supported without fetching the actual objects. This is seen from Line 14 where the term frequency of the new object is compared with the lowest term frequency of already existing object representations in the cell. Therefore, we do not need to fetch already inserted objects in order to maintain the $targetedK$ most relevant objects of a cell.

4.3 Query Processing

We proceed by giving an exact query processing algorithm for the TkSTR query. The query takes 3 arguments: a region, a keyword, and the desired number of result objects. In order to use the bit strings for query processing, we first encode the query region to match a relevant grid cell. Any cell describes at most $targetedK$ objects except the finest-level cells. For each face of the cube, the query region may be equal to or smaller than a grid cell, which yields three different scenarios:

1. $Q.k$ or more of the objects described by the cell are in the query region.
2. Fewer than $Q.k$ of the objects described by the cell are in the query region, and the cell contains fewer than $targetedK$ objects.
3. Fewer than $Q.k$ objects described by the cell are in the query region, and the cell contains $targetedK$ objects.

When $Q.k$ objects are described by the cell and they are all in the query region, they can be returned as the result because the objects described by the cell are sorted by term frequency and no object can be more relevant. With fewer than $targetedK$ objects in the cell, the objects in the query region can be returned as a result because no more objects can exists at any lower level. However, with $targetedK$ objects in the cell, more objects may exist at lower levels, and an exact result cannot be provided without considering lower cells.

Since a cell that covers the query region provides an exact result in the first two scenarios, we initially encode the query region as the smallest grid cell that covers it. More cells have to be examined when the initial cell cannot provide a result, as seen from the third scenario. The query processing procedure is described in Algorithm 2. The first two scenarios are covered by Lines 5–9. When the result set contains $Q.k$ objects, the loop in Line 10 is not entered because there are no candidate cells, and the result set is returned. With a result set smaller than $Q.k$, the result is only returned if the cell contains fewer than $targetedK$ objects.

The third scenario is covered by Line 10. In each iteration, the cells from the lower level are fetched until no better object can exist in lower cells. Only cells from the next level that are overlapping $Q.R$ are examined, as seen in Line 13. Lower levels for cells that contain $targetedK$ objects may have to be fetched, as shown in Line 17. Cells with fewer than $targetedK$ objects cannot have lower cells with new objects and are therefore not refined. Thus, dense cells with a large number of objects may be examined to the finest level, whereas lower levels of sparse cells may not be fetched.

All objects in $Q.R$ are added to the result set in Line 16. Notice that the result set is ordered and maintains only $Q.k$ objects. Lower levels are fetched for cells with $targetedK$ objects when the result set contains fewer than $Q.k$ objects. When the result set contains $Q.k$ objects, there may exist better objects in a lower cell. Consider the example in Figure 2, where the most relevant object is outside the query region. In the case where the $targetedK$ most relevant objects are outside the query region, there may still exist objects that can enter the result set. Therefore, lower levels are fetched until no better object can exists in the query region, as shown in Line 20, or until the lowest level has been examined.

The query processing algorithm iterates through all objects in a cell before continuing to the next cell in the candidate set. The $Q.k$ most relevant objects may be found be combining the first few objects of a number of cells. In this special case, Algorithm 2 has a computational overhead of examining objects that are not relevant in order to provide the result. The algorithm may be modified to process the candidate cell in a parallel fashion similar to how the Threshold Algorithm [9] functions. With this approach, the cell with the most relevant objects is considered first, resulting in fewer computations. The pseudo-code is omitted due to space limitations.

Example 4. Consider the two queries Q_1 and Q_2 with $Q_1.k = 2, Q_1.keyword = "pizza", Q_2.k = 2, Q_2.keyword = "shoe"$ and with the query regions in Figure 7: $Q_1.R = R_1$ and $Q_2.R = R_2$. We first run Algorithm 2 using the database from Table 3 and query Q_1:
Fetching key: *00000*
R: (3,4)
No object can exists in any cell with a better term frequency since $Q.k$ objects are inside the query region (Line 8). The result of the algorithm is (3,4).
Next, we run the algorithm with the same settings for the query Q_2:

Algorithm 2: `STRQuery`(Database DB, Query Q)

```
   /* Global variables: */
1  targetedK ← max obj. ref. stored in DB;          // conf. param.
   /* Local variables: */
2  R ← result set with Q.k objects ordered by term frequency;
3  minCell ← smallest cell that covers Q.R ;
4  candCells ← empty set of candidate cells ;
5  foreach Object o ∈ minCell_{Q.keyword} do
       // objects are sorted - most relevant first
6      if o is inside Q.R then
7          R ← R ∪ o ;                          // can only be the best
8  if |R| < Q.k ∧ |minCell_{Q.keyword}| = targetedK then
9      candCells ← minCell ;                    // more objects may exists
10 while candCells ≠ ∅ do
11     lowerCells ← empty set of cells ;
12     foreach cCell ∈ candCells do
13         foreach cell inside cCell ∧ overlapping Q.R do
14             foreach Object o ∈ cell_{Q.keyword} do
15                 if o is inside Q.R then
16                     R ← R ∪ o ;
17             if |cell_{Q.keyword}| = targetedK then
                   // more objects may exist
18                 if |R| < Q.k then
                       // provide Q.k objects
19                     lowerCells ← lowerCells ∪ cell ;
20                 else if cell_{Q.keyword}.mintf > R[Q.k].tf then
                       // something better may exist
21                     lowerCells ← lowerCells ∪ cell ;
22     candCells ← lowerCells ;
23 return R ;
```

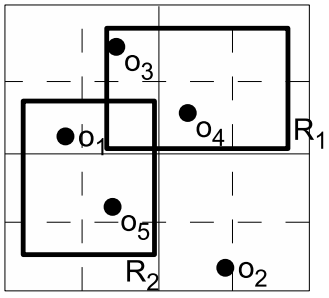


Figure 7: Querying.

Fetching key: *00010*
R: (5) *(object 2 is outside $Q.R$)*
Since the result set is smaller than $Q.k$ and the cell contains $targetedK$ objects, more relevant objects may exist (Line 8).
Fetching keys: *0000010* and *0000110* (overlap with $Q.R$)
R: (5,1)
Both cells have less than $targetedK$ cells. Thus, no more objects may exist (Line 17). The result of the algorithm is (5,1). □

5. EXPERIMENTAL EVALUATION

We proceed to evaluate the proposed solution. We first present the experimental setup and then describe the datasets. Finally, we report on a number of experiments.

5.1 Experimental Setup

All experiments are performed on a commodity machine with a 64-bit quad-core Intel i5-2520M (2.5 GHz) processor with 8 GB of main memory. We use a standard installation of PostgreSQL version 9.3.4 with an 8 KB page size. All data is loaded into a table with two columns (key, value), where the key column is set to be primary key. The primary key is indexed as default with a B-tree. The storage mode of the value column is set to *external* to prevent compression when the data cannot fit into an 8 KB page. The proposed solution is implemented in a single-threaded Java application that executes SQL commands using JDBC for database communications. The data is read from disk, and no data is maintained in main memory by the Java application. We employ the default hashing function provided by the Java Framework.

We perform two sets of experiments. The first reports on the properties of the data stored in the PostgreSQL database when varying $targetedK$, the number of grid levels, and the hash value length using two datasets. The second evaluates the query processing algorithm when varying on the query arguments, dataset, and storage parameters.

5.2 Datasets

We collected all world-wide geo-tagged messages from the public Twitter Streaming API during February 2013. For this study, we create two datasets from these messages. The first dataset (termed GPCAT) simulates Google Places by filtering the Twitter messages with the 95 categories used by Google Places[1]. The first 100,000 messages that contain any of the category names in their textual description are added to the dataset GPCAT. The resulting dataset contains messages with 47 of the categories. Thus, the number of terms to encode to a bit string is 47. Since many users make use of services such as Foursquare that automatically reports on the location of a user, e.g., at restaurants, parks, and airports, this dataset may reflect many of the actual POIs provided by Google Places.

The second dataset (termed FULL) is created by taking the first 100,000 messages from the Twitter dataset without filtering the textual descriptions. This dataset contains textual descriptions in any language containing any terms. In total, it contains 53,247 unique terms.

With dataset GPCAT, we may see a large number of bit string collisions since the messages are filtered by 95 categories in English. The dataset FULL may provide a large number of unique bit strings since terms may exist in any language from anywhere in the world. Therefore, the two different datasets are used to explore the properties of the indexing and query processing when the data varies.

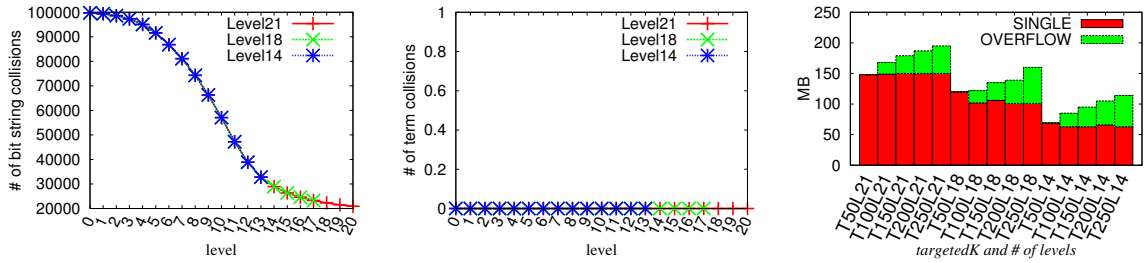
5.3 Storing Objects

In the first set of experiments, we evaluate the performance of the proposed storage and indexing technique. Also, we report on the properties of the stored data. We vary different aspects in the experiments.

5.3.1 Varying the Number of Grid Levels

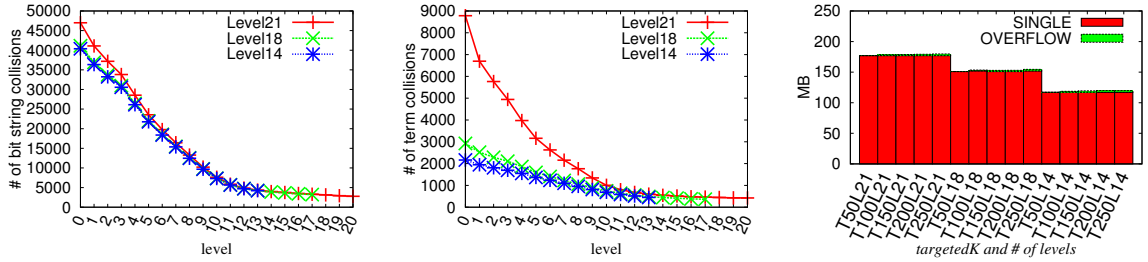
We aim at storing data such that reasonable sized query regions are supported efficiently. The number of grid levels to consider in encoding the bit string influences the effectiveness. Therefore, we vary on the number of maximum grid levels such that we store both datasets GPCAT and FULL with a maximum number of grid levels that results in the finest cell being approximately 10, 100, and 1,000 meters. This corresponds to grid levels 21, 18, and 14, which are

[1]http://developers.google.com/places/documentation/supported_types



(a) Number of Bit String Collisions **(b)** Number of Term Collisions **(c)** Storage Requirements

Figure 8: Storage Statistics for the GPCAT dataset



(a) Number of Bit String Collisions **(b)** Number of Term Collisions **(c)** Storage Requirements

Figure 9: Storage Statistics for the FULL dataset

represented by 45, 39, and 31 bits, respectively.

5.3.2 *Varying the Hash Value Length*

The dataset GPCAT contains many objects that share terms, which may result in bit string collisions. The amount of bit string collisions vary with the length of the hash value. We propose to fit all bit strings into 64 bits, which matches the supported memory addresses of our CPU. With the above-mentioned maximum grid levels, we have 19, 25, and 33 bits available for the textual part of the bit string. This provides a reasonable number of possible hash values considering the properties of the two datasets.

5.3.3 *Varying* $targetedK$

The parameter $targetedK$, which also depends on the setting of $Q.k$, determines the number of objects to store for each term in each cell, and it influences the effectiveness of the query processing algorithm. We store the two datasets with $targetedK$ set to 50, 100, 150, 200, and 250.

5.3.4 *Collisions*

In the first experiments, we insert the datasets into the DBMS and report on the number of collisions. When a bit string representation of the location and the term of an object is to be stored, it may already exists since another object in the same cell may contain the same term. The larger the cell, the more likely it is that another object exists with the same term. This is seen from Figures 8a and 9a that show the number of bit string collisions in both datasets.

For dataset GPCAT that contains only a few terms, we see almost 100,000 collisions at grid level 0 because the objects with the same term in each of the cells have the same bit string representation. As the cells becomes smaller, the number of collisions decreases for both datasets. Notice that each of the three maximum levels has a different hash value length, but for dataset GPCAT, the levels have the same number of collisions. This is because GPCAT contains very few terms, resulting in no term collisions as seen in Figure 8b. This means that each unique term results in a unique hash value for all three hash value lengths.

Dataset FULL produces different numbers of bit string collisions for the three maximum levels since the length of the hash value varies. As seen in Figure 9a, there are many more collisions when the hash value is 19 bits compared to 25 and 33 bits. This is because the number of unique hash values increases with the number of bits available. Figure 9b describes the number of term collisions for dataset FULL, i.e., when different terms are hashed to the same value. With 25 bits hash values (maximum grid level 18), we already see a relatively low number of term collisions compared to the number of bit string collisions. Therefore, increasing the number of bits for the textual part of the bit string does not provide a large improvement in the number of term collisions for these settings.

Dataset GPCAT contains many more bit string collisions than dataset FULL, as seen in Figures 8a and 9a, because of the textual filtering. The filtering enforces a limited number of English terms, and therefore the objects are more likely to be located in same parts of the world compared to dataset FULL.

5.3.5 *Storage Requirements*

The object representations stored for each bit string may occupy more space than the page size of 8 KB. Figures 8c and 9c report on the space consumption for dataset GPCAT and FULL. The lower part of the bars reports on the amount of storage necessary to store data that fits inside a single page. The upper part of the bars reports on the amount of data that did not fit into a single page. As expected, less data in total is stored for small values of $targetedK$ and for a low number of levels. With large values of $targetedK$, we see an increase in overflown pages for dataset GPCAT because more objects are stored for each term. Dataset FULL creates very few overflown pages because there are few bit string collisions. Dataset GPCAT has many collisions, which results in more overflown pages. As seen from Figure 8c, the number of overflown pages increases with $targetedK$. This occurs because more objects have to be stored in each cell.

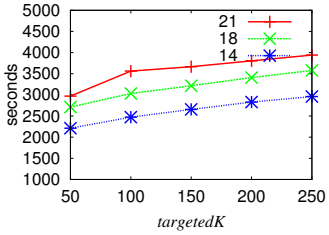
The number of tuples required to store the datasets at each of the maximum grid levels is listed in Table 4. The dataset FULL contains many more tuples than dataset GPCAT because it has fewer collisions. This increases the number of unique bit strings.

Dataset	Level 21	Level 18	Level 14
GPCAT	860,655	628,138	333,766
FULL	1,783,060	1,516,271	1,133,870

Table 4: Number of Tuples for the Datasets

5.3.6 *Storage Time*

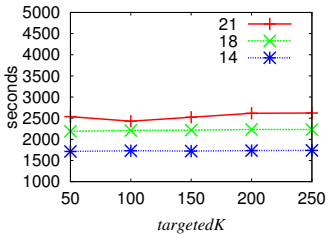
In this experiment, we examine the time to store the objects. As seen from Figures 10 and 11, the storage time increases with the number of levels to store; this is because more tuples have to be created. In Figure 10, we also see an increase in the storage time for dataset GPCAT when $targetedK$ is increased; this is because it takes more time to store the additional objects. The dataset produces a large number of collisions, resulting in more objects to store for each bit string.



Level	Index Size (MB)
21	35
18	25
14	13

Fig. 10 & Table 4: Storage Time and Index Size for GPCAT

In Figure 11, the storage time for dataset FULL is more constant because there are fewer collisions, making it less sensitive to the size of $targetedK$. Overall, it is faster to process the FULL dataset even though it contains more tuples. This is due to the fewer collisions that increase the time to sort and update the order of the most relevant objects.



Level	Index Size (MB)
21	54
18	46
14	34

Fig. 11 & Table 5: Storage Time and Index Size for FULL

5.3.7 *Index Size*

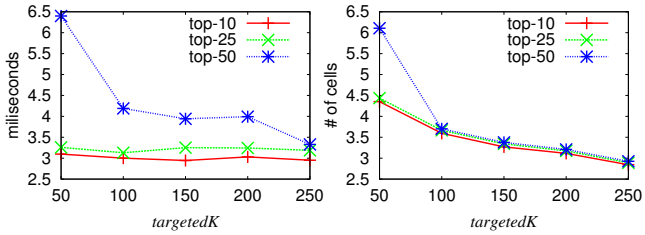
The size of the indexes does not vary with the value of $targetedK$ because only the bit strings are indexed. Therefore, only the number of bit strings to index influences the size of the index, as seen from Tables 4 and 5. The index sizes for the two datasets follow the number of rows seen in Table 3, as expected.

5.4 Query Processing

In this set of experiments, we study the performance of the query processing algorithm. We employ both datasets and design two query sets for each dataset. For both query sets, we create 1,000 queries for each parameter value, and we vary $Q.k$ and $targetedK$. Also, we randomly choose a term for $Q.keyword$ from the bag of all terms. Thereby, we query frequent terms more often. We randomly choose a query region that may have a size between that of grid level 0 and 1 cm^2.

5.4.1 *Dataset GPCAT*

For the dataset GPCAT, we make all query regions cover New York City, USA in order to ensure we use a region with a large number of objects that contain terms from the Google Places categories. As seen in Figure 12a, the query runtime increases when $Q.k$ is close to $targetedK$; this occurs because more cells have to be fetched to produce a result. The number of fetched cells is shown in Figure 12b, where the number of cells required to answer the query decreases when $targetedK$ increases. The query runtime increases as expected with $Q.k$ because more objects have to be examined. When the value of $Q.k$ equals $targetedK$, the query runtime increases because more cells have to be fetched in order to produce a result.

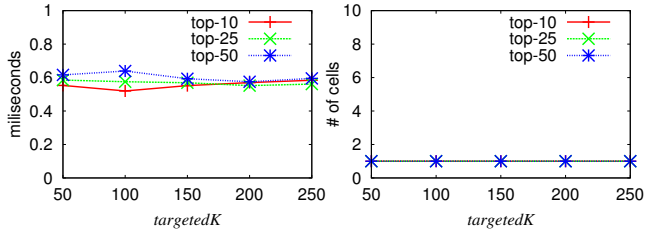


(a) Query Runtime **(b)** Number of Fetched Cells

Figure 12: Query Runtime for Dataset GPCAT

5.4.2 *Dataset FULL*

We allow the query region for the dataset FULL to be located anywhere in the world. As seen from Figure 13a, the query runtime varies little regardless of the values of $Q.k$ and $targetedK$. Only a single cell lookup is required for all queries, as seen in Figure 13b. This is either because of the few collisions that result in cells with fewer than $targetedK$ objects stored for most bit strings or because the query region may find an empty region.



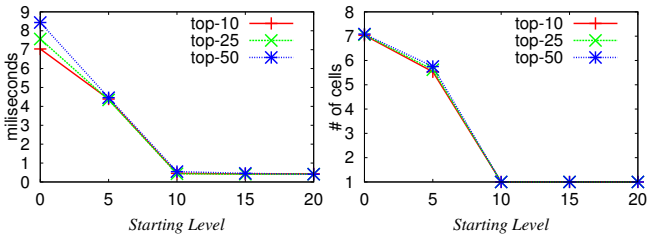
(a) Query Runtime **(b)** Number of Fetched Cells

Figure 13: Query Runtime for Dataset FULL

5.4.3 *Varying Maximum Number of Levels*

In the next experiment, we study dataset GPCAT and use the same query sets as before, but fix $targetedK$ at 250 and vary the starting grid level. We force a query to start at a given grid level and evaluate the runtime and number of fetched cells. As seen in Figure 14a, the query runtime increases when the starting grid level approaches 0. This is because of the large number of collisions at these grid levels, as seen in Figure 8a. The same applies to the number of fetched cells, as seen in Figure 14b. Almost the same number of cells is fetched when varying $Q.k$ because $targetedK$ is sufficiently large. Therefore, even though $Q.k$ varies, the query may be answered with the same number of cells. This was also verified in the experiment shown in Figure 12b. There is a small computation overhead when $Q.k$ increases because more objects must be examined; this is seen in Figure 14a. At grid level 10, it is

seen that a single cell is sufficient to answer the queries since few collisions exists at this grid level. We omit coverage of the same study for dataset FULL because the result can be returned with a single cell at any grid level, as seen in Figure 13b.



(a) Query Runtime **(b)** Number of Fetched Cells

Figure 14: Query Runtime for Dataset GPCAT

6. CONCLUSIONS AND RESEARCH DIRECTIONS

We present a new framework for the processing of top-k spatial keyword queries, which take a keyword and a query region as arguments, against sets of spatio-textual objects. The framework leverages standard indexes that are available in DBMSs and key-value stores, and it requires no new index structures for its operation. Specifically, spatio-textual objects are encoded as compact bit strings that may be stored and indexed using a standard index such as a B-tree. The spatial part of a bit string is obtained using an existing, multi-layered grid-structure, and the textual part is obtained using hashing. To reduce the storage requirements, only a limited number of objects are stored in each intermediate grid cell. Query processing works by combining objects from multiple grid levels until an exact result can be returned.

An experimental study with real data offers insight into the properties of the stored objects and the performance of query processing. The study demonstrates that the framework is capable of efficiently offering query results using an existing DBMS, and for different datasets.

Several directions for continued research exist:

Support for queries with more than one keyword. While supporting single keywords in some sense generalizes approaches that only support retrieval based on categories, it is relevant to also support multiple keywords. Solutions that "combine" the results of single keyword queries or ones that combine terms in bit strings may be explored.

Support for deletion. Several directions may be considered, including tombstoning deleted objects until a given threshold is reached, at which point the objects must be deleted physically. Deletions may be accommodated by fetching object from lower cells until $targetedK$ objects are again maintained.

Experiments with larger datasets. It is of interest to see how the proposed solution perform with even more data.

Acknowledgments

This research was supported in part by the MADALGO research center, in part by the Geocrowd Initial Training Network, funded by the European Commission as an FP7 Peoples Marie Curie Action under grant agreement number 264994, and by a grant from the Obel Family Foundation.

7. REFERENCES

[1] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972.

[2] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD*, pages 322–331, 1990.

[3] A. Cary, O. Wolfson, and N. Rishe. Efficient and scalable method for processing top-k spatial Boolean queries. In *SSDBM*, pages 87–95, 2010.

[4] L. Chen, G. Cong, C. S. Jensen, and D. Wu. Spatial keyword query processing: An experimental evaluation. *PVLDB*, 6(3):217–228, 2013.

[5] Y.-Y. Chen, T. Suel, and A. Markowetz. Efficient query processing in geographic web search engines. In *SIGMOD*, pages 277–288, 2006.

[6] M. Christoforaki, J. He, C. Dimopoulos, A. Markowetz, and T. Suel. Text vs. space: Efficient geo-search query processing. In *CIKM*, pages 423–432, 2011.

[7] G. Cong, C. S. Jensen, and D. Wu. Efficient retrieval of the top-k most relevant spatial web objects. *PVLDB*, 2(1):337–348, 2009.

[8] I. De Felipe, V. Hristidis, and N. Rishe. Keyword search on spatial databases. In *ICDE*, pages 656–665, 2008.

[9] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *In PODS*, 102–113, 2001.

[10] R. Finkel and J. Bentley. Quad trees a data structure for retrieval on composite keys. *Acta Informatica*, 4(1):1–9, 1974.

[11] Google Inc. Google S2 Geometry Library. `http://code.google.com/p/s2-geometry-library/`, 2011. [Online; accessed September 2014].

[12] A. Guttman. R-trees: A dynamic index structure for spatial searching. *SIGMOD*, pages 47–57, 1984.

[13] R. Hariharan, B. Hore, C. Li, and S. Mehrotra. Processing spatial-keyword (sk) queries in geographic information retrieval (gir) systems. In *SSBDM*, pages 16–16, 2007.

[14] D. Hilbert. Uber die stetige abbildung einer linie auf ein flachenstuck. In *Mathematische Annalen, 38*: 459-460, 1891.

[15] A. Khodaei, C. Shahabi, and C. Li. Hybrid indexing and seamless ranking of spatial and textual features of web documents. In *DEXA*, pages 450–466, 2010.

[16] D. E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*, 2nd ed., Addison-Wesley. 1998.

[17] Z. Li, K. Lee, B. Zheng, W.-C. Lee, D. L. Lee, and X. Wang. IR-tree: An efficient index for geographic document search. *TKDE*, 23(4):585–599, 2011.

[18] J. a. B. Rocha-Junior, O. Gkorgkas, S. Jonassen, and K. Nørvåg. Efficient processing of top-k spatial keyword queries. In *SSTD*, pages 205–222, 2011.

[19] S. Vaid, C. B. Jones, H. Joho, and M. Sanderson. Spatio-textual indexing for geographical search on the web. In *SSTD*, pages 218–235, 2005.

[20] D. Wu, G. Cong, and C. S. Jensen. A framework for efficient spatial web object retrieval. *VLDBJ*, 21(6):797–822, 2012.

[21] D. Wu, M. L. Yiu, G. Cong, and C. S. Jensen. Joint top-k spatial keyword query processing. *TKDE*, 24(10):1889–1903, 2012.

[22] Y. Zhou, X. Xie, C. Wang, Y. Gong, and W.-Y. Ma. Hybrid index structures for location-based web search. In *CIKM*, pages 155–162, 2005.

[23] V. Gaede, and O. Günther. Multidimensional Access Methods. In *CSUR*, pages 170-231, 1998