Text vs. Space: Efficient Geo-Search Query Processing

Maria Christoforaki
Computer Science & Eng.
Polytechnic Institute of NYU
christom@cis.poly.edu

Jinru He
Computer Science & Eng.
Polytechnic Institute of NYU
jhe@cis.poly.edu

Constantinos Dimopoulos
Computer Science & Eng.
Polytechnic Institute of NYU
constantinos@cis.poly.edu

Alexander Markowetz
Computer Science
University of Bonn, Germany
alex@iai.uni-bonn.de

Torsten Suel
Computer Science & Eng.
Polytechnic Institute of NYU
suel@poly.edu

ABSTRACT

Many web search services allow users to constrain text queries to a geographic location (e.g., yoga classes near Santa Monica). Important examples include local search engines such as Google Local and location-based search services for smart phones. Several research groups have studied the efficient execution of queries mixing text and geography; their approaches usually combine inverted lists with a spatial access method such as an R-tree or space-filling curve. In this paper, we take a fresh look at this problem. We feel that previous work has often focused on the spatial aspect at the expense of performance considerations in text processing, such as inverted index access, compression, and caching. We describe new and existing approaches and discuss their different perspectives. We then compare their performance in extensive experiments on large document collections. Our results indicate that a query processor that combines state-of-the-art text processing techniques with a simple coarse-grained spatial structure can outperform existing approaches by up to two orders of magnitude. In fact, even a naïve approach that first uses a simple inverted index and then filters out any documents outside the query range outperforms many previous methods.

Categories and Subject Descriptors

H.3.3 [**INFORMATION STORAGE AND RETRIEVAL**]: Information Search and Retrieval; H.3.4 [**INFORMATION STORAGE AND RETRIEVAL**]: Systems and Software

General Terms

Algorithms, Experimentation, Performance

Keywords

Efficient query processing, Geographic web search engines

CIKM'11, October 24–28, 2011, Glasgow, Scotland, UK.

1. INTRODUCTION

In just a few years, *geographic web search* has emerged from a niche service to one of the most popular applications. It allows users to focus queries to a particular geographic region; e.g., "yoga lessons near Los Angeles" to yoga schools in the LA area, or "pizza" to pizza stores near the user's current location (as reported by her smartphone). The underlying data collections usually either originate from business directories (yellow pages), or consist of *geo-coded* web documents. In the latter case, the search provider found a *geographic reference* (e.g., a city name, or an address) in a document, and was able to map this to a location. As data on the web grows in size, more and more detailed local information becomes available, making geographic web search more useful. In addition to the increase in geo-coded documents, the increasing use of GPS-enabled smartphones further boosts the number of geographic queries.

This massive amount of geographic queries necessitates efficient query processing. Commercial search providers maintain vast server farms to process this workload, and even a small increase in efficiency results in significant savings. However, geographic queries differ significantly from traditional text queries. Standard inverted indexes hence do not directly extend to geographic search, nor do the existing techniques for query optimization. The main challenge lies in the combination of textual and spatial constraints, i.e., finding pages that contain query terms *and* are close to the intended location. Both textual and spatial queries have been extensively studied within their respective communities, but much less work has been spent on combining them.

On the one hand, research in *Information Retrieval* (IR) has resulted in extremely fast algorithms for textual search using inverted indexes. In particular, major web search engines employ a range of performance optimizations including caching, index compression, and early termination. For typical setups, these techniques are able to execute most (or all) queries directly from main memory, running in just a few milliseconds. The rare cache misses only lead to sequential disk access, still significantly cheaper than random I/O. On the other hand, research in spatial databases has proposed numerous specialized index structures such as quad-trees or R*-trees. These however commonly result in a fair amount of random I/O, once the data exceeds main memory. This behavior is partially due to (geo-)database systems having different objectives than web search systems. However, it

raises the question of how to best integrate spatial structures in geographic web search.

Over the past years, several studies have explored techniques for efficient processing of geographic queries over document collections. While there are a number of different algorithms, most of them rely on a single R-tree, into which the textual index entries are then inserted. The resulting index structure contains a tremendous number of small textual index structures at the leaves of the R-tree. Significant amounts of CPU are hence spent navigating the spatial structure, even after employing various pruning heuristics. Moreover, when the data does not fit in main memory, this approach necessitates multiple random I/Os per query, rendering it too expensive for web search applications.

We observe that in any typical geo web search scenario, textual clearly dominates spatial data. In particular, the average web page contains several hundred distinct terms, but only one or few geographic references. Even yellow page entries commonly contain at least tens of terms, but only a single location (the address of the business). Any efficient method for processing geographic web queries thus has to primarily address text indexing. Namely, it has to encorporate the various performance optimizations employed in current web search engines. Conversely, a solution that focuses on organizing documents into a fine-grained spatial structure results in significantly decreased performance.

In this paper, we substantiate this claim by experimentally comparing three basic approaches: First, a *Naïve R*-Tree* maintains an inverted index at each leaf, indexing all documents within the leaf's MBR. Second, a *Clairvoyant R*-Tree* assumes an oracle that can prune unproductive subtrees, and thus supersedes many of the optimizations in the literature. Third, a brute-force *Text-First* baseline first determines all textually relevant documents using a state-of-the-art inverted index implementation. Only thereafter, it discards documents outside the query area. Our experimental results show that *Text-First* beats both R*-tree methods, and substantially outperforms the numbers reported in previous papers.

Of course, we should expect a geographic search query to run faster than the Text-First approach, since it actually only needs to be evaluated on a subset of "local" documents. We thus further optimize our approach, by integrating fairly coarse-grained, and thus light-weight, spatial structures into the inverted index without adversely impacting its performance. This leads us to algorithms based on kd-trees and space-filling curves that outperform the CPU cost of our baseline by up to two orders of magnitude.

In our experimental evaluation, we try to create setups that closely resemble scenarios encountered by commercial geographic search services. Thus, we use documents and queries extracted from real data sets that preserve the natural correlation between text and geographic location. In addition, we consider cases where (i) data is completely in main memory, (ii) data is on disk but partially cached in main memory, and (iii) data is on disk with no caching. The latter case is in fact not realistic in current engines, which all have at least a substantial cache of index data in main memory. In fact, cache hit rates commonly range at 90% or higher, due to the natural skew in query term frequencies. We also evaluate the algorithms on additional synthetic data sets, in order to observe the impact of various parameters. Overall, our contributions can be summarized as follows:

- We show that a brute-force inverted index-based method without any spatial index structure outperforms the previous techniques from the literature.
- We describe, implement, and evaluate several optimizations, integrating a coarse-grained spatial structure into the inverted index. The proposed approaches achieve significant speed-ups over the baseline method, and thus over previous work.
- We evaluate the various methods through extensive experiments on real and synthetic data.

The remainder of the paper is structured as follows. First, Section 2 provides background and discusses previous work on text indexing, spatial data structures, and geographic IR. Next, Section 3 outlines the basic application scenario and describes the data setup. Section 4 presents the baseline approaches and performs a brief experimental comparison of these methods. Then, Section 5 discusses how to further optimize performance using coarse kd-trees and space-filling curves. Section 6 evaluates the various approaches and optimizations through a large set of experiments. Finally, Section 7 concludes the paper.

2. BACKGROUND AND RELATED WORK

Geographical search engine technology builds on contributions from several research communities, in particular (i) text indexing and query processing, (ii) spatial index structures, and (iii) geographical information retrieval. In the following, we give a short overview of relevant techniques in these areas and subsequently discuss previous work on query processing in geographic search queries.

2.1 Indexing and Querying in Search Engines

There are a number of textual index structures in the literature including inverted indexes [28], signature files [9], and suffix arrays [18, 1]. However, almost all current web search engines and text information retrieval systems are based on inverted indexes, and substantial efforts have been invested in optimizing the construction, size, and access speed of these structures; see, e.g., [28, 26].

An *inverted index* contains one *inverted list* for each distinct term (word) in the document collection. Each list consists of postings describing occurrences of the term in the collection. We assume that a posting contains the ID of the document where the term occurs (docID), and the number of times it occurs (frequency). Postings may contain additional data such as the precise positions of the terms in the document. However, docIDs and frequencies already suffice to compute simple common ranking functions such as BM25 or Cosine measures. The postings in each list are usually sorted by docID, and an additional dictionary structure stores for each distinct term a pointer to the start of the corresponding list. Query processing basically involves intersecting (conjunctive queries) or merging (disjunctive queries) the inverted lists of the query terms and then computing the ranking function on top. The main performance challenge is the length of the inverted lists for common query terms, which increases with collection size.

While inverted indexes and basic query mechanisms seem almost trivial, there are a number of non-trivial optimizations such as index compression, caching, index pruning and

reorganization, early termination, and parallel query processing. These can increase query throughput drastically [28]. As a result, state-of-the-art search tools achieve impressive query processing, typically in the range of a few milliseconds per query, on machines indexing millions of pages. We refer to [26, 22] for recent numbers on the widely used GOV2 collection of 25 million pages. As we demonstrate in this paper, such state-of-the-art techniques for text query processing are crucial for building high-performance geographic search engines.

2.2 Spatial Indexing

There exists a tremendous body of work in spatial databases and geographic information systems on index structures for spatial data. The main approaches can be classified into: (i) tree-based methods [8], (ii) grid structures [20], and (iii) space-filling curves [12, 3, 21]. Most relevant to our work here are tree-based methods and space-filling curves.

Tree-based indexing structures can be divided into (i) space- and (ii) data-partitioning. The former divides space into disjoint tiles, while the latter divides the spatial objects into disjoint subsets. Important examples of space-partitioning techniques are quad-trees [11] and kd-trees [2]. Their major drawback is that objects that span across a border between tiles are stored twice. This problem occurs for relatively large objects, but not when objects are fairly small or just points. The most widely used data-partitioning approach is the R-tree family of index structures [13]. These store each object only once, but use overlapping subareas that may result in a search operation having to visit multiple subtrees. An optimized version, the R*-tree, employs an improved partitioning heuristic, and is widely used in many applications. We refer to [8] for a more detailed discussion of tree-based spatial index structures.

Space-filling curves (SFCs) are continuous curves that cover the entire space. The basic goal of a SFC is that most points that are close to each other in space should also be reasonably close to each other on the curve. There are a number of different SFCs in the literature, with the Z curve and Hilbert curve being the most widely used. For an overview of space-filling curves, we refer to [12, 3, 21]. In this paper, we use index structures for geo search based on R*-trees, kd-trees, and space-filling curves.

2.3 GIR and Geo Query Processing

Geographic information retrieval (GIR) deals with items (documents) containing both spatial data and unstructured data, most commonly text. We refer to [16] for early work, and to [15] for a recent overview. This area commonly requires techniques from both textual information retrieval and spatial data processing.

We focus on efficient query processing over web documents and yellow page collections under both textual and spatial constraints, as motivated by the local and mobile search services provided by all the major search engines. Thus, we have a set of documents, each containing textual content and one or more locations (points or rectangles). Given a query consisting of terms and a rectangle, the goal is to find (and possibly rank) all documents that contain the query terms and have a location in the query rectangle.[1]

[1]There are several variations of this problem, but for now this basic version suffices.

A number of researchers have studied this and related problems over the last few years, starting with the early work in [23], where three techniques based on a grid scheme for spatial indexing along with inverted indexes are proposed. In the first approach, an inverted index is built for each grid cell and each query is performed by first applying spatial filtering and then textual. In their second technique, the postings are spatially clustered based on the grid cells their footprints belong to. The third approach separately applies spatial and textual filtering and merges the results. This work is the most closely related to our work. The main drawback of the second proposed approach, which is similar to our work, is that the documents in lists do not preserve a sorted order which in turn does not permit performing effective skipping while processing the inverted lists.

Some follow-up work was done in [27, 6]. In particular, Zhou et al. in [27] studies three approaches combining inverted indexes and R*-trees: (i) keeping separate indexes, (ii) an inverted file on top of an R*-tree (the overall best), and (iii) an R*-tree on top of an inverted file.

The work by Chen et al. in [6] combines inverted indexes with space-filling curves. More precisely, efficient query processing is achieved by laying out the inverted lists along a space-filling curve. This work is also different from [27] as it focuses on disk performance and assumes that larger spatial payloads (not just points or rectangles) may be attached to the documents, and that a document may relate to several locations. However, the CPU performance in [6] suffers from bottlenecks in the inverted index implementation.

Several more recent papers have proposed additional query processing algorithms based on combining inverted indexes with R*-trees [14, 10, 7, 17]. These papers all use R*-trees as the main structure, and basically insert the document data into the R*-tree. Basic query processing involves traversing the R*-tree to visit leaf nodes satisfying the spatial contraint, and then traversing the (fairly small) textual indexes in the leaves. Additional improvements are obtained by pushing some textual information up in the tree, allowing pruning of subtrees that do not contain the query terms or cannot contain high-scoring results.

In particular, in the KR*-tree algorithm [14], dictionary information is propagated from leaves to internal nodes. In the IR^2-tree algorithm in [10], R*-trees are combined with signature files [10]. Thus, every node in the IR^2-tree maintains a signature of the textual content in its subtree. Cong et al. [7] look at a slight variation of the problem where only the top-k results need to be returned, based on textual relevance and spatial proximity (but without an a-priori cut-off distance as implied by a query rectangle). They build an optimized R*-tree structure for this problem that also stores suitable summary information in the internal nodes. Follow-up work in [4] studies a modified ranking function based on textual relevance, proximity, and a new prestige measure derived from query logs. Finally, the work in [17] presents a structure called IR-Tree that stores impact and location information for selected high-scoring terms in intermediate nodes of the R*-tree, enabling additional pruning during top-k query processing.

In this paper, we revisit the problem of efficient geo query processing, with emphasis on both CPU time and disk performance. Our goal is to design techniques that integrate state-of-the-art techniques from IR query processing, and that achieve significant improvements over existing approaches.

In our experiments we will show that a fairly general class of approaches based on R*-trees cannot come close to our best techniques in the most commonly studied scenarios.

3. PROBLEM AND EXPERIMENT SETUP

We assume conditions similar to those encountered by commercial geo search engines such as Google Local. Specifically, we envision a scenario where entities have a precise location; e.g., a web page about a cafe that is assigned the business' street address. Thus, each document in our collection is associated with a single point location. (Though, we could extend this to multiple locations or small areas with negligible modifications in the code.) In the remainder of the paper, we refer to these markers as the *geographic footprint* of a document. Each query consists of several keywords, and a rectangular geographic search area. The latter is usually small, at least compared to the continental U.S., the region under investigation. We refer to this set of rectangles as *query footprints*.

We pursue a realistic geographic web search scenario, similar to that encountered by a large local (geographic) search engine. For a realistic setting, it is of utmost importance to use data sets that reflect the natural relationship between (i) terms and positions and (ii) keywords and documents. For example, a page about opera tickets is more likely to relate to a large city; the query "cattle feed" most likely originates from a rural location.

To maintain this natural relationship between terms and locations, we geo-coded real web documents and search queries. To this end, we scanned each document for geographic markers, such as zip codes or names of counties, cities, and towns. Next, we assigned a geographic position to the document, using the "Census 2000 U.S. Gazetteer" [5]. This data collection maps $95,000$ geographical entities (zip codes and location names) to geographic coordinates. In cases where we found more than one location in a document, we chose the one with the larger population. For example, if a document contained a reference to New York City as well as Smalltown, we assumed it belonged to the former. Similarly, we geo-coded web queries. Namely, we first parsed real web queries for geographic terms, subsequently translated into a geographic position. Finally, we associated each query with a search rectangle, corresponding to the square miles of the geographical location found.

We conduct our experiments on two sets of documents. First, we use 6.1 million pages from a broad web crawl performed by our group (referred to as $R6.1$). These documents were assigned locations by means of geo-coding, as described above. They were part of a larger collection, from which we removed all web pages that did not refer to a U.S. location. Second, we use the 25 million documents of the $GOV2$ data set (referred to as $F25$). Here we assign random locations, however according to the distribution observed in $R6.1$. The characteristics of the $R6.1$ web data collection summarize as follows. There are a total of 6.1 million geo documents, containing $29,872,888$ distinct terms. The size of the uncompressed inverted index is 17.9 GB, whereas its compressed version is only 3.36 GB. Our collection contains $14,134$ distinct geographic footprints. The $F25$ data set consists of $25,205,179$ documents, $36,759,149$ distinct words and $6,797$ M postings. We also obtained much larger data sets of size 50, 75 and 100 million documents by suitably replicating the $F25$ data set.

A set of geographic web queries was obtained by geo-coding entries of the 2006 AOL search log. In particular, we selected queries containing the name of a town, county, or a zip code. The query was then assigned a search rectangle centered around this location.Consequently, the term indicating the geographic position (e.g., the name of a town) was removed from the query string. The resulting query trace contains $49,978$ geo queries, $5,841$ distinct locations and an average of 2.8 textual terms per query.

We do not assume any particular search scenario, as could arise from a person on foot vs. a person travelling by car (larger area of interest). To model different scenarios, we partition our geo queries into four subsets, called *small*, *medium*, *large*, and *mixed*, based on the sizes of their footprints. In particular, the small set corresponds to areas of less than 0.5 sq. miles, the medium set to areas between 0.5 and 450 sq. miles and the large set to areas larger than 450 sq. miles. For every experiment we used $1,000$ queries from each set.

We compute the precise relevance scores for all documents, but allow the choice of any k in top-k pruning approaches. We focus on conjunctive queries and retrieve all the documents that satisfy the textual and geographical constraints of the query. To be more specific, we consider every document that (i) has a non-zero term-based score, and (ii) whose spatial focus lies strictly within the query rectangle. For the purpose of this paper, we use a simple scoring function. In particular, we employ a linear combination of a term-based measure (BM25) and a geographic score, the distance between the center of the query footprint and the geographic footprint of the document. However, our approach also applies to many other ranking functions.

4. BASELINES AND PRELIMINARIES

In this section, we first describe several basic solutions, and then show experimentally that a simple *Text-First* index already outperforms previous approaches. Finally, we discuss these observations, providing valuable insights for the subsequent optimizations.

4.1 The Naïve R*-Tree Algorithm

As described in Section 2, a number of previous approaches have used R(*)-trees. We describe the *Naïve R*-Tree*, the most basic such approach. This method inserts documents into an R*-Tree according to their geographic locations. At each leaf node, there is a small inverted index over all documents residing therein. Given a query, we first traverse the R*-tree to find all leaf nodes intersecting the query footprint. Subsequently, we access the inverted indexes stored at each of these leaves. The approach thus first filters by space, and only thereafter by text. We implemented the Naïve R*-Tree by adding inverted indexes to the leaf nodes of the canonical R*-tree implementation of [19].

4.2 The Clairvoyant R*-Tree Algorithm

While the Naïve R*-Tree does not perform any textual filtering before reaching the leaf level, several authors have proposed textual pruning at intermediate nodes. We hence conceptualize a *Clairvoyant R*-Tree*, employing perfect textual pruning inside the R*-Tree, at zero cost. This hypothetical data structure would thus outperform any of the approaches using R*-trees, as presented in the literature.

The Clairvoyant R*-Tree structurally resembles a Naïve R*-Tree, but employs an oracle that indicates if a path leads

to a leaf containing results. Recall that while traversing the Naïve R*-Tree we may visit leaves that do not contain a single result. For example, consider the query "cat, dog", on the index structure of Figure 1. Assume that the query rectangle intersects spatially with all depicted leaf nodes. The intermediate nodes of the Naïve R*-Tree only store spatial information. One would thus visit all leaves, and traverse the inverted lists for "cat" and "dog" in nodes n_3 and n_4. However, only n_4 actually returns a result, and exploring n_3 wastes considerable resources, due to R*-tree traversal, dictionary lookup, and inverted index access. In contrast, the Clairvoyant R*-Tree features an oracle at each intermediate node of the R*-tree. For a given query, it directs us exclusively to leaves in the subtree that contain at least one result. The oracle thus allows us to avoid any nodes in the tree that are not on a path between the root and leaves containing results. In our example, it would spare us from visiting n_3 as well as exploring this leaf's inverted index, while still finding the results at n_4.

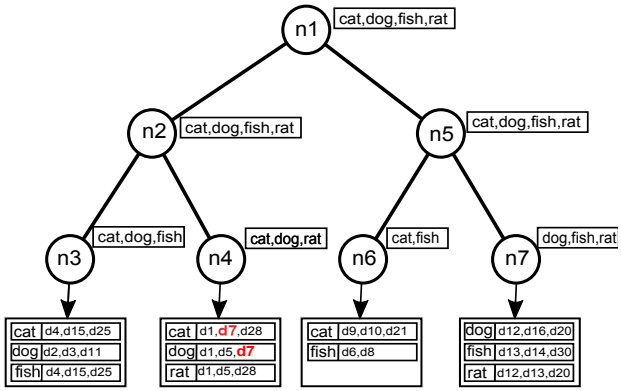

Figure 1: A hybrid R-Tree based structure.

Obviously, this data structure is entirely hypothetical. Yet, it makes an excellent contestant for our experiments, as it subsumes all optimizations storing textual hints at intermediate nodes. In fact, it subsumes many optimizations not in the literature. We simulated its implementation by executing every query twice. First, we issue an exploratory query, recording all inner nodes that actually lead to a result. Subsequently, we execute the actual query, only visiting these productive nodes.

The main limitation of the Clairvoyant R*-Tree is that the algorithm must visit and score every document that satisfies the textual and spatial filters. Some top-k early termination techniques such as in [7, 17] however do not score every document passing the filters. The performance of these data structures is not bounded by the Clairvoyant R*-Tree. We further elaborate on this topic below.

4.3 Text-First Baseline

We now describe a trivial algorithm that relies on an inverted index, and uses no spatial index structure at all. While this algorithm is trivial, we are not aware of any previous work that compares an optimized implementation of it to the various R-Tree based methods. More precisely, we use an inverted index structure, and a simple array storing the geo locations of the documents. This array, almost two orders of magnitude smaller than the inverted index, fits into main memory. This *Text-First* algorithm then executes a query as follows:

1. Issue a query against the inverted index, retrieving the docID of any document containing all query terms.
2. For each docID returned, perform a lookup in the array to check if the document's location is in the query interval.
3. For any docID passing this spatial check, fetch the frequency values of the postings and compute the BM25 score.

In our implementation, we used an inverted index that compresses docIDs and frequencies in blocks of 128 elements using the OPT-PFD algorithm [25]. We executed queries on our own *document-at-a-time* (DAAT) query processor, optimized through block-wise compression and forward skips in the inverted lists. DocIDs were assigned to documents at random; thus we did not try to obtain additional benefits due to sorted assignment as described in [25]. Overall, this implementation contains many of the techniques used in state-of-the-art IR query processors. To illustrate its performance, we later also report experimental results on this text index alone (*Text-Only*), without applying any geographic filtering at all.

4.4 Preliminary Experimental Evaluation

We implemented the baseline methods and conducted experiments using the *R*6.1 data set and the mixed query set. Table 1 shows the CPU costs of the above methods in milliseconds when both spatial and textual indexes reside in memory. Observe that the trivial Text-First method clearly outperforms both R*-tree methods. The Clairvoyant R*-Tree is much faster than the Naïve R*-Tree, but still significantly slower than Text-First. This illustrates the importance of comparing newly proposed spatial optimizations to simple baselines employing state-of-the-art IR query processing techniques. Comparing Text-First and Text-Only, the former performs additional table lookups and intersections with the query rectangle. This spatial check however results in fewer frequency values being fetched and BM25 scores computed. The performance of both algorithms is thus very similar. Note that to guarantee fairness during this preliminary evaluation, both spatial and textual index structures are memory-resident.

Method	time/query
Naïve R*-Tree	33.0ms
Clairvoyant R*-Tree	12.0ms
Text-Only	7.2ms
Text-First	6.9ms

Table 1: CPU costs of baseline approaches (R6.1)

We now compare the CPUs costs to those reported in previous papers. The best published results appear to be those in [7], where queries on 2.5 million web pages take about 20 ms of CPU time on a similar processor, while our results are for 6.1 million pages. Thus, even though our Clairvoyant R*-Tree does not provide a bound for the top-k methods in [7], as discussed earlier, all our numbers compare well to their reported CPU numbers. In fact, [7] also provides numbers for larger data sets up to 10 million pages, but performance degrades in this case. We note that results reported in other previous papers are limited to even smaller data sets, and achieve much slower performance relative to data size.

In conclusion, it is important to integrate state-of-the-art IR query processing into geographic search. Moreover, for several million pages, R-tree based methods do not appear to perform as well as a carefully implemented trivial baseline. Note that these findings are not restricted to R(*)-trees in particular; any other deep spatial data structure (e.g., kd-trees) would have shown similar performance. All these indexes spend significant amounts of time navigating the fine-grained spatial structure and performing the multiple dictionary lookups to find the inverted lists in the leaf nodes. The Text-First baseline algorithm does not have these overheads.

Of course, when further increasing data size to tens and hundreds of millions of pages, the trivial baseline will eventually fail. In the next section, we thus add coarse-grained spatial optimizations that achieve significant improvements. We evaluate these algorithms on data sets up to 100 million pages, assuming data to be entirely in memory, entirely on disk, and cached in memory. This scenario thus exceeds any experimental setup in literature.

5. IMPROVED ALGORITHMS

As seen in the previous section, even the simple Text-First approach outperforms the best results in the literature in terms of CPU performance. In this section, we present various improved algorithms with optimizations, which as we will see through the experimental results in Section 6 perform significantly better than the baseline solutions. In fact, we outline and compare two approaches and their variations: (i) a coarse-space partitioning (CSP) and (ii) a spatial indexing based on a space-filling curve (SFC).

5.1 Coarse Space Partitioning

The first improved algorithm uses a fairly coarse-grained partitioning of the geographic space into a limited number of regions. Subsequently, it runs Text-First on any region intersecting the query (typically just one, as regions are larger than most query rectangles). We would thus like to divide space into a small number of k regions such that each contains approximately the same number of documents. One could think of various optimizations that try to avoid, e.g., cutting through major metropolitan areas, but we decided to partition space using a simple kd-tree. We hence recursively divide space along vertical and horizontal axes, until the desired number of regions has been obtained. The optimal choice of k naturally depends on the size of the collection and the sizes of the query footprints, a trade-off explored thereafter. Choosing a relatively small value of k allows us to fit the dictionary structures of all regions into main memory, avoiding random I/O at query time. (Caching is also known to perform extremely well on disk-based dictionaries, due to the extreme skew in query term frequencies). We obtain the following simple query processing algorithm:

1. Determine all regions that intersect the query rectangle, using either a scan over an array or a traversal of the (small) kd-tree structure.
2. Run Text-First in each intersecting region, fetching list data from disk if needed.

We implemented two versions of this method, using different layouts of inverted lists. In the first approach, called *CSP-SEQ*, the k inverted lists for each term are stored sequentially. They are thus organized as a single inverted list with the k dictionaries pointing to the starts of the k sublists. In the second approach, *CSP-SUB*, each region builds its own index substructure independently. As seen later, the choice of layout does not significantly affects CPU time. However, when data resides on disk, CSP-SEQ tends to outperform CSP-SUB. Its sequential layout commonly consumes only a single disk seek, even when several regions intersect with the query rectangle. Only in rare cases do lists get so long that it would be preferable to utilize two or three separate seeks. In our memory-resident experiments we present only the results of the CSP-SEQ algorithm for simplicity, since the choice of layout doesn't affect CPU time of the coarse-space partitioning methods.

5.2 Space-Filling Curves

This approach uses space-filling curves [12] to lay out index structures in geo search engines. In fact, as recently disclosed in [24], all the major search engines currently use index structures based on space-filling curves in their local search services. Thus, our results can be seen as providing experimental support that such methods indeed outperform methods based on other, more involved, spatial data structures. Its core idea was proposed by [6], albeit under a very different setup. The authors of [6] assumed that each location may have an associated *document footprint record* of up to several hundred bytes. These may contain polygon data or textual annotations that needs to be fetched at query runtime. Space-filling curves were thus utilized to organize these footprints on disk. Also, the work in [6] did not employ a state-of-the-art IR query processor, and thus reports higher CPU costs.

We use a so-called Z curve [21], which is a fairly simple but still effective SFC. (The approach extends directly to other types of space-filling curves, such as Hilbert.) We first assign to each document a docID that corresponds to the position of its location on the SFC, and then build a standard block-compressed inverted index. The idea is that all documents in the query rectangle are likely close to each other on the SFC, and thus close to each other in the inverted list. This allows skipping large parts of the list. To achieve this skipping effect, we explore two approaches described in the following.

SFC-QUAD: The first approach uses a quad-tree structure to enable skipping. We built a fairly shallow and thus small quad-tree over the document space, roughly consuming 0.5 MB. We then first traverse the quad-tree to determine up to m docID *ranges* that contain all documents intersecting the query footprint. The query processor, adapted again from the Text-First approach, hence only accesses (decompresses) these m docID ranges and can skip the rest of the inverted lists. Moreover, when data resides on secondary storage, the query processor only retrieves the relevant ranges of the inverted lists from disk.

We evaluated different values of m, and found that in main memory, values in the order of several hundred perform best. When fetching inverted list from disk, such a fine-grained approach fails, since for any given list, one should spend only a few (often just one, and rarely more than three) random seeks. We thus merge the m ranges into a smaller number of k disk *sweeps*. In fact, we can choose the number of sweeps optimally, based on the lists lengths and layouts as well as disk model. Overall, we obtain the following algorithm:

1. Traverse the quad-tree and obtain m docID ranges covering all documents in the query rectangle.
2. For each term t whose inverted list is not in main memory, determine $k = 1$, 2, or 3 docID ranges. These should cover the above m docID ranges and minimize total I/O cost.
3. For each such term perform k seeks on disk to fetch the needed parts of the inverted list.
4. Run Text-First on the m docID ranges.

SFC-SKIP: We also explored an alternative, yet natural, way to skip parts of the docID space. Remember that our inverted lists are compressed in blocks of 128 postings. Suppose that for each such block we store a *minimum bounding rectangle* (MBR), encompassing all of the locations of documents represented in the block. During query processing, we can skip any block whose MBR does not intersect with the query rectangle. This skipping fits naturally into any typical IR query processor, which already performs skips over compressed blocks of postings. In fact, we can store additional MBRs for groups of, say, 2, 4, 8, etc. blocks. The resulting structure thus allows hierarchical skipping. This structure differs from the others approaches presented in this paper, because it is defined on a per-list basis. Thus, we have more detailed skipping information for longer lists, and no such information at all for lists with fewer than 128 postings.

We explored different settings and found that storing MBRs and shortcut pointers on 3 levels, for 4, 16, and 64 compressed blocks, provided excellent performance. The MBRs and shortcuts are stored separately from the lists and are usually kept in main memory. Using this information, we can again determine optimized sweeps for fetching index data from disk, as in the SFC-QUAD method. In our disk-resident experiments we found that the performance of the two versions of the SFC approach is similar. Thus, in our experiments when data reside on disk, we only present the performance of SFC-QUAD for simplicity.

6. EXPERIMENTS

In this section we present the results of an extensive performance evaluation of our proposed algorithms.

6.1 Experimental Evaluation

In Section 4, preliminary experiments showed that the Text-First *baseline* outperforms the commonly considered R-Tree based approaches in terms of CPU. In this section, we investigate the benefits achieved by the various optimizations from Section 5. In particular, we evaluate three approaches: (i) *Coarse Space Partitioning* (CSP), (ii) *Space Filling Curve Quad Tree* (SFC-QUAD), and (iii) *Space Filling Curve Skip pointers* (SFC-SKIP). The parameters under investigation are: (a) the size of the query rectangle, (b) the number k of sweeps, (c) the amount of memory available for caching index data, and (d) the size of the data set. By default, we used query rectangles of mixed size, and a choice of up to three sweeps. In every set of experiments, we modify one parameter, while the others remain at their default. All experiments were conducted on a single core of an Intel Xeon server with $2.27Ghz$.

We first conduct experiments in main memory. Subsequently, we evaluate setups where data resides on secondary storage. In particular, we use a standard disk model to estimate the time spent retrieving data from disk. We assume $8ms$ access time for each random access, and $50MB/s$ sequential transfer rate, settings typical for current SATA disks. In the remainder of the paper, we refer to the total disk cost including both seek and transfer time as I/O cost. For CPU results all structures are kept in main memory, while for disk and disk-cached results, only the spatial structures plus the stated amount of cached inverted lists are in memory while the rest of the inverted list is on disk.

Index size: Table 2 reports the space required to store the inverted index with docIDs and term frequencies. We index (and compress) gaps between docIDs, so smaller gaps among docIDs achieve better compression. We observe that CSP has larger index size than other methods, because docIDs are scattered into separate indexes (partitions). The relative gap between docIDs of the same partition hence increases, resulting in a larger docID index. On the other hand, the baseline and SFC approach achieve smaller index size due to their smarter docID assignment scheme. Of course, the increased index size for CSP also has an adverse effect on disk access costs (both with and without caching).

in GB	docID	freq	total
Baseline	2.49	0.87	3.36
CSP	4.55	1.12	5.67
SFC	2.50	0.87	3.37

Table 2: Compressed index size (R6.1)

Spatial structure size: Table 3 presents the size of the spatial structures in memory. We observe that the R*-Tree is significantly larger than the CSP and SFC methods. For each document we assign footprints of 8 bytes and thus need roughly 48 MB of memory for maintaining this information for the R6.1 data set. The size of these footprints is not included in any of the spatial structure sizes reported, and is thus listed separately. Moreover, we assume that the document footprints always reside in memory.

	size in MB
R*-tree	680.935
CSP-SEQ	0.004
SFC-QUAD	0.513
SFC-SKIP	17.502
Doc footprints	48.8

Table 3: Size of spatial structures (R6.1)

Query footprint size: First, we vary the size of the query rectangles. Table 4 illustrates the effects of this parameter on CPU consumption, using the R6.1 data set. We observe that all methods significantly improve over the Text-First baseline. These results confirm our expectation that coarse spatial partitioning outperforms any fine grained partitioning. Overall, light-weight spatial indexing techniques such as SFC achieve best performance. The spatial ordering of documents in the inverted lists according to SFC answers even queries with large footprint in less than one millisecond. For this setting, we also evaluated different index layout for the CSP method. Results however indicated that the choice of layout in main memory does not noticeably affect performance. Next, we evaluate the proposed methods on disk-resident data. In Table 5 we show experimental results, assuming that the inverted index resides entirely on disk and

in ms	mixed	small	medium	large
Baseline	6.95	6.81	6.97	7.10
CSP-SEQ	0.50	0.14	0.28	1.27
SFC-QUAD	0.14	0.07	0.08	0.40
SFC-SKIP	0.17	0.08	0.09	0.48

Table 4: Query footprint size vs. CPU (R6.1)

using the R6.1 data set. Note that the CSP-SEQ and SFC approaches achieve significantly better performance than the baseline method. The SFC-QUAD method clearly dominates all other approaches. The experimental results allow the following observations. First, we observe the same ordering of the algorithms for I/O cost as for CPU cost. Second, the layout of inverted lists on disk significantly affects efficiency. In particular, the CSP-SEQ method with sequential layout outperforms its rival CSP-SUB using smaller independent inverted indexes. It is thus beneficial to write all inverted lists for the same term sequentially on disk. Queries located close to a partition border can thus be answered with a single seek operation per term.

in ms	mixed	small	medium	large
Baseline	79.1	79.1	79.1	79.1
CSP-SEQ	29.1	24.8	27.3	44.2
CSP-SUB	41.2	34.5	40.2	62.9
SFC-QUAD	25.1	23.6	24.9	34.1

Table 5: Query footprint size vs. I/O (R6.1)

For this parameter, we also investigate seek and transfer time individually. Table 6 presents the transfer time for various query footprint sizes. In contrast, seek time remains stable between $22ms$ to $28ms$ for all methods and settings. This is not surprising, if we recall that queries on average contain 2.8 terms. For each term, we need to perform at least one seek (of about $10ms$), and thus expect a total seek time of $28ms$. Only in the case of large footprints and the CSP-SUB method, seek time increases to $43.2ms$. This is expected, since large footprints are more likely to cross partition borders, thus necessitating multiple random seeks for the same term.

For the Text-First baseline, transfer cost strongly dominates seek time, since the method ignores spatial filtering, and thus frequently retrieves redundant blocks. Note that the baseline's transfer time remains the same for different footprint sizes, since inverted lists are always retrieved in full length; the same holds for the seek time. All proposed optimizations achieve notably lower transfer cost than the baseline, because the spatial filtering enables them to retrieve fewer blocks. Also, note that the layout of inverted lists on disk affects transfer cost. In particular, sequential layout shows smaller transfer cost for small query footprints. As footprints become larger, both layouts start to achieve similar transfer cost.

in ms	mixed	small	medium	large
Baseline	56.7	56.7	56.7	56.7
CSP-SEQ	5.9	3.1	4.7	17.8
CSP-SUB	13.2	10.5	12.2	19.7
SFC-QUAD	2.3	1.2	2.1	10.1

Table 6: Query footprint vs. disk transfer (R6.1)

Number of sweeps: Next, we evaluate the degree to which the number of sweeps affects performance. Table 7 depicts I/O cost as the number of sweeps increases. Since SFC-QUAD and SFC-SKIP perform exactly the same I/O, we only report the numbers of the former. In the *normal* setting, the algorithm performs exactly k sweeps. Using the *hybrid k-sweeps* technique, the optimum of up to k sweeps is chosen. In particular, it is beneficial to choose a large k for long inverted lists. As these become longer, multiple sweeps improve performance by skipping redundant blocks. In both data sets we observe that, as the number of sweeps increases, performance drops significantly. On the R6.1 data set, hybrid methods exhibit stable performance, very close to 1-sweep. On the F25 data set, the option of using larger k in the hybrid approach yields more substantial benefits. This effect can largely be explained by F25 having longer inverted lists than R6.1.

	$R6.1$		$F25$	
in ms	normal	hybrid	normal	hybrid
1 sweep	26.5	26.5	41.4	41.4
2 sweeps	58.1	25.6	68.6	36.8
3 sweeps	79.6	25.1	93.4	35.2

Table 7: Effects of the number of sweeps in SFC

Cache size: So far, we have assumed that the index structure is either completely in main memory, or only on disk. However, the latter case is not realistic. In real large-scale local search engines the index is either entirely in memory or a significant fraction of the index is cached. To determine the impact of caching, we apply a simple scheme that maintains a static subset of lists in memory. Inverted lists are cached according to (i) their frequency in a large query trace, and (ii) their length. Caching small lists is slightly more preferable, due to the high cost of random accesses on disk. As shown in Figure 2, the I/O cost decreases significantly as cache size grows. Allowing $512MB$ of cache, the total disk cost for SFC-QUAD decreases to less than 10% of the cost without caching. Also, we observe that the relative performance gap between SFC-QUAD versus coarse-grained methods increases with cache size.

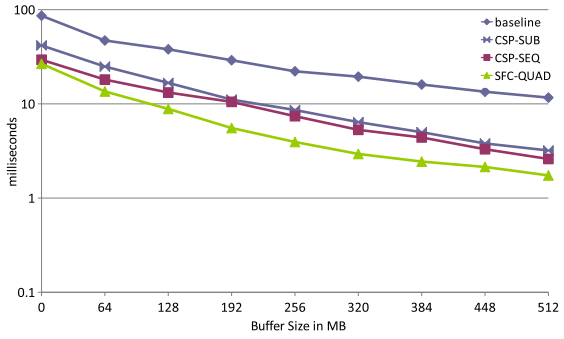

Figure 2: Effects of cache size (R6.1)

Data set size: Table 8 shows the CPU costs (SFC-Quad and SFC-Skip) and I/O-costs (Hybrid k-sweeps) when evaluating the SFC methods for document collections of various sizes. Namely, we use 50, 75 and 100 million documents. The total cost is the sum of the CPU and I/O costs. We observe that all optimizations scale extremely well to larger data. Also, note that the quad-tree approach consistently outperforms skip pointers.

Partition size: Next, we explore the optimal number of partitions under the CSP approach. Again, we experiment in main memory as well as on disk. Coarse partitioning is

in ms	6.1	25	50	75	100
SFC-QUAD	0.14	0.23	0.55	0.75	0.89
SFC-SKIP	0.17	0.24	0.57	0.81	0.93
Hybrid k-sweeps	25.1	29.7	31.2	32.8	34.4

Table 8: Number of documents vs. performance

tested by scaling the number of partitions (of the continental US) from 8 to 1024. As expected, the optimal partition granularity differs between main memory and disk. In Figure 3, we observe the effect of partition granularity in main memory for different query footprint sizes. Initially, as the number of partitions increases, performance also improves. Note that with 256 partitions, performance for different query footprint sizes converges to optimal. Increasing the number of partitions does not further reduces the cost of query processing.

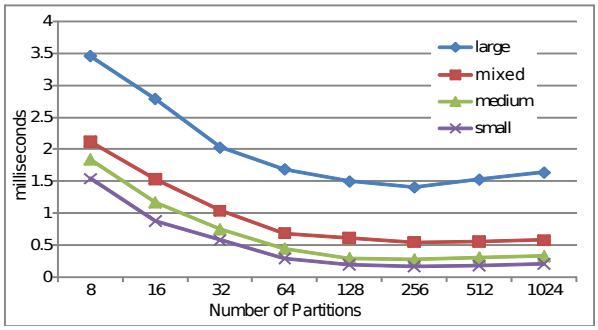

Figure 3: Partition granularity vs. CPU (R6.1)

Figure 4 depicts the impact of the number of partitions on I/O cost. Overall, we observe that as the number of partitions increases, performance degrades. This is reasonable, since more partitions intersect with each query on average, resulting to an increase in the number of disk seeks for fetching all inverted lists. When partition size is set to 64, we observe that all algorithms achieve optimal performance. Further fine grained partitioning leads to additional disk overhead. This experiment supports our claim that fine grained partitioning does not further improve performance. In terms of I/O efficiency, this setting encounters a trade-off between transferring large inverted lists and performing multiple seeks. The best choice for the number of partitions naturally depends on the size of the collection and the sizes of the query footprints.

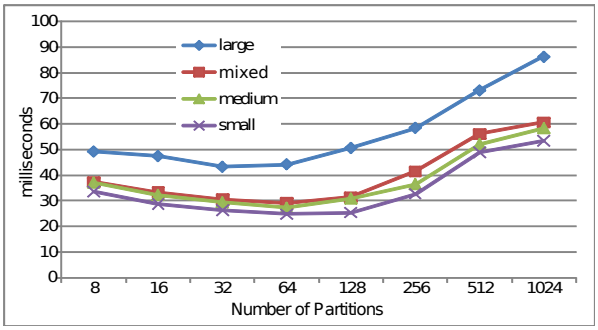

Figure 4: Partition granularity vs. I/O (R6.1)

Query terms: We also tested the effect of the number of query terms on the performance in main memory, using the R6.1 data set. According to Table 9, the CSP-SEQ and SFC methods show significant improvements over the baseline. The SFC approaches outperform CSP-SEQ, while SFC-QUAD performs slightly better than SFC-SKIP. The performance of all algorithms decreases as the number of the query terms increases, due to more lists being accessed.

in ms	1	2	3	4 or 5	≥ 6	mean
Baseline	4.01	6.71	6.92	8.21	8.43	6.95
CSP-SEQ	0.43	1.23	1.53	1.91	2.43	1.51
SFC-QUAD	0.07	0.12	0.16	0.21	0.27	0.14
SFC-SKIP	0.07	0.13	0.16	0.23	0.31	0.17

Table 9: Query length vs. CPU (R6.1)

Document size: Next, we study how CPU performance changes depending on the average number of distinct terms in each document. Table 10 shows the impact on the query time for our methods, as we utilize 100, 50 and 30% of the text, meaning the percentage of random distinct terms per document taken into account. More text naturally results in longer inverted lists. Therefore, more blocks must be retrieved which leads to additional performance overhead. CSP-SEQ and SFC methods achieve much better response time than our baseline. SFC approaches achieve remarkably smaller time than other proposed methods. The relative order between the different algorithms is not changed by varying the document size.

in ms	100%	50%	30%
Baseline	6.95	3.21	2.01
CSP-SEQ	0.51	0.33	0.21
SFC-QUAD	0.14	0.07	0.05
SFC-SKIP	0.17	0.10	0.07

Table 10: Document size vs. CPU (R6.1)

Query footprint size ($F25$): We also evaluate the impact of the size of the query footprint. In particular, we compare the performance of our proposed methods when tested on the $F25$ data set. Furthermore, we evaluate query performance when data resides (i) in memory as well as (ii) on disk. The SFC approaches once again perform best, across all query footprint sizes. As seen in Table 11, they outperform their competitors in terms of CPU cost, even on large document collections. Also, the CPU cost increases with the size of query footprints, since more inverted list data needs to be retrieved.

in ms	mixed	small	medium	large
Baseline	9.78	9.53	9.61	9.98
CSP-SEQ	1.05	0.35	0.63	3.97
SFC-QUAD	0.23	0.13	0.15	0.60
SFC-SKIP	0.24	0.14	0.17	0.65

Table 11: Query footprint size vs. CPU (F25)

According to Table 12, our proposed methods also outperform the baseline on disk. Query footprints naturally affect performance, since larger query rectangles intersect with more documents.

7. DISCUSSION AND CONCLUSIONS

This paper studied the query processing problem that arises in geographic search services such as local search and search on mobile devices. We described several new and existing algorithms, and experimentally evaluated their performance on real and synthetic data sets. Our best methods achieve very substantial improvements over all previous results.

in *ms*	mixed	small	medium	large
Baseline	122.72	122.72	122.72	122.72
CSP-SEQ	37.13	34.59	36.14	56.78
CSP-SUB	46.32	41.33	47.78	74.61
SFC-QUAD	29.7	28.4	31.5	39.7

Table 12: Query footprint size vs. I/O (F25)

The main lessons to be taken away from our results are as follows:

1. State-of-the-art query processing techniques for text data are quite efficient. For related problems such as geographic web search, it is thus important to consider approaches that preserve and exploit these techniques.

2. As there is significantly more textual than spatial data in current geo search engines, it is important to focus on the textual aspect of the problem.

3. Approaches based on fine-grained spatial structures may result in significant CPU and I/O overheads compared to more coarse-grained structures.

4. Efficient algorithms must avoid random I/O, since a single random I/O is significantly more expensive than the entire CPU cost of a query on millions of pages.

5. The best approaches use a careful layout of the inverted lists (and corresponding assignment of docIDs) according to spatial structure.

6. Since current search engines cache a substantial amount (or all) of the index data in memory, focusing on the disk-only case is not sufficient.

There are of course still many open problems in geographic search. Additional optimizations and other methods are an obvious research direction. It would also be interesting to look at parallel query processing in geo search engines, where data is distributed over many nodes. Also, in this paper we focused on simple ranking functions such as BM25, while current engines use several ranking phases that apply increasingly sophisticated ranking functions. Finally, there are many open problems on aspects of geographic search other than efficiency.

Acknowledgments

This research was supported by NSF Grant IIS-0803605, "Efficient and Effective Search Services over Archival Webs".

8. REFERENCES

[1] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addision Wesley, 1999.

[2] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[3] C. Böhm, G. Klump, and H.-P. Kriegel. XZ-Ordering: A Space-Filling Curve for Objects with Spatial Extension. In *SSD*, pages 75–90, 1999.

[4] X. Cao, G. Cong, and C. S. Jensen. Retrieving top-k prestige-based relevant spatial web objects. *PVLDB*, 3(1):373–384, 2010.

[5] Census 2000 U.S. Gazetteer. `http://www.census.gov/geo/www/gazetteer/places2k.html`.

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

[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] M. de Berg, M. van Krefeld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer, 2000.

[9] C. Faloutsos and S. Christodoulakis. Signature files: An access method for documents and its analytical performance evaluation. *ACM Trans. Inf. Syst.*, 2(4):267–288, 1984.

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

[11] R. A. Finkel and J. L. Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Inf.*, 4:1–9, 1974.

[12] V. Gaede and O. Günther. Multidimensional access methods. *ACM Comput. Surv.*, 30(2):170–231, 1998.

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

[14] R. Hariharan, B. Hore, C. Li, and S. Mehrotra. Processing spatial-keyword (SK) queries in geographic information retrieval (GIR) systems. In *SSDBM*, page 16, 2007.

[15] C. B. Jones and R. S. Purves. Geographical information retrieval. *International Journal of Geographical Information Science*, 22(3):219–228, 2008.

[16] R. Larson. Geographic information retrieval and spatial browsing. In *GIS and Libraries: Patrons, Maps and Spatial Information*, pages 81–124, 1996.

[17] Z. Li, K. C. K. Lee, B. Zheng, W.-C. Lee, D. L. Lee, and X. Wang. IR-Tree: An Efficient Index for Geographic Document Search. *IEEE Trans. Knowl. Data Eng.*, 23(4):585–599, 2011.

[18] U. Manber and E. W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.

[19] Marios Hadjieleftheriou. Spatial Index Library. `http://www2.research.att.com/~marioh/spatialindex/`.

[20] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Trans. Database Syst.*, 9(1):38–71, 1984.

[21] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann, 2006.

[22] T. Strohman and W. B. Croft. Efficient document retrieval in main memory. In *SIGIR*, pages 175–182, 2007.

[23] 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.

[24] P. Venetis, H. Gonzalez, C. S. Jensen, and A. Y. Halevy. Hyper-local, directions-based ranking of places. *PVLDB*, 4(5):290–301, 2011.

[25] H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. In *WWW*, pages 401–410, 2009.

[26] J. Zhang, X. Long, and T. Suel. Performance of compressed inverted list caching in search engines. In *WWW*, pages 387–396, 2008.

[27] 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.

[28] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2), 2006.