# Efficient Query Processing in Geographic Web Search Engines

Yen-Yu Chen
Polytechnic University
Brooklyn, NY 11201, USA
yenyu@photon.poly.edu

Torsten Suel
Polytechnic University
Brooklyn, NY 11201, USA
suel@poly.edu

Alexander Markowetz
Hong Kong University of Science and Technology
Hong Kong, S.A.R
alexmar@cs.ust.hk

## ABSTRACT

Geographic web search engines allow users to constrain and order search results in an intuitive manner by focusing a query on a particular geographic region. Geographic search technology, also called *local search*, has recently received significant interest from major search engine companies. Academic research in this area has focused primarily on techniques for extracting geographic knowledge from the web. In this paper, we study the problem of efficient query processing in scalable geographic search engines. Query processing is a major bottleneck in standard web search engines, and the main reason for the thousands of machines used by the major engines. Geographic search engine query processing is different in that it requires a combination of text and spatial data processing techniques. We propose several algorithms for efficient query processing in geographic search engines, integrate them into an existing web search query processor, and evaluate them on large sets of real data and query traces.

## 1. INTRODUCTION

The World-Wide Web has reached a size where it is becoming increasingly challenging to satisfy certain information needs. While search engines are still able to index a reasonable subset of the (surface) web, the pages a user is really looking for are often buried under hundreds of thousands of less interesting results. Thus, search engine users are in danger of drowning in information. Adding additional terms to standard keyword searches often fails to narrow down results in the desired direction. A natural approach is to add advanced features that allow users to express other constraints or preferences in an intuitive manner, resulting in the desired documents to be returned among the first results. In fact, search engines have added a variety of such features, often under a special *advanced search* interface, but mostly limited to fairly simple conditions on domain, link structure, or modification date.

In this paper we focus on geographic web search engines, which allow users to constrain web queries to certain geographic areas. In many cases, users are interested in information with geographic constraints, such as local businesses, locally relevant news items, or tourism information about a particular region. For example, when searching for yoga classes, local yoga schools are of much higher interest than the web sites of the world's largest yoga schools.

We expect that *geographic search engines*, i.e., search engines that support geographic preferences, will have a major impact on search technology and their business models. First, geographic search engines provide a very useful tool. They allow users to express in a single query what might take multiple queries with a standard search engine. A user of a standard search engine looking for a yoga school in or close to Brooklyn, New York, might have to try queries such as

- `yoga ''new york''`
- `yoga brooklyn`
- `yoga ''park slope''` (a part of Brooklyn)

but this might yield inferior results as there are many ways to refer to a particular area, and since a purely text-based engine has no notion of geographical closeness (e.g., a result across the bridge to Manhattan or nearby in Queens might also be acceptable). Second, geographic search is a fundamental enabling technology for *location-based services*, including electronic commerce via cellular phones and other mobile devices. Third, geographic search supports locally targeted web advertising, thus attracting advertisement budgets of small businesses with a local focus. Other opportunities arise from mining geographic properties of the web, e.g., for market research and competitive intelligence.

Given these opportunities, it comes as no surprise that over the last two years leading search engine companies such as *Google* and *Yahoo* have made significant efforts to deploy their own versions of geographic web search. There has also been some work by the academic research community, e.g., [32, 8, 12, 1, 19, 29, 30], mainly on the problem of extracting geographic knowledge from web pages and queries. Our approach here is based on a setup for geographic query processing that we recently introduced in [30] in the context of a geographic search engine prototype. While there are many different ways to formalize the query processing problem in geographic search engines, we believe that our approach results in a very general framework that can capture many scenarios.

### 1.1 Problem Statement and Motivation

We focus on the efficiency of query processing in geographic search engines, e.g., how to maximize the query throughput for a given problem size and amount of hardware. Query processing is the major performance bottleneck in current standard web search engines, and the main reason behind the thousands of machines used


*SIGMOD 2006,* June 27–29, 2006, Chicago, Illinois, USA.

by larger commercial players. Adding geographic constraints to search queries results in additional challenges during query execution which we now briefly outline.

In a nutshell, given a user query consisting of several keywords, a standard search engine ranks the pages in its collection in terms of their relevance to the keywords. This is done by using a text index structure called an inverted index to retrieve the IDs of pages containing the keywords, and then evaluating a term-based ranking function on these pages to determine the $k$ highest-scoring pages. (Other factors such as hyperlink structure and user behavior are also often used, as discussed later). Query processing is highly optimized to exploit the properties of inverted index structures, stored in an optimized compressed format, fetched from disk using efficient scan operations, and cached in main memory.

In contrast, a query to a geographic search engine consists of keywords and the geographic area that interests the user, called *query footprint* [30]. Each page in the search engine also has a geographic area of relevance associated with it, called the *geographic footprint* of the page. This area of relevance can be obtained by analyzing the collection in a preprocessing step that extracts geographic information, such as city names, addresses, or references to landmarks, from the pages and then maps these to positions using external geographic databases. In other approaches it is assumed that this information is provided via meta tags or by third parties. The resulting page footprint is an arbitrary, possibly noncontiguous area, with an amplitude value specifying the degree of relevance of each location. Footprints can be represented as polygons or bitmap-based structures; details of the representation are not important here.

A geo search engine computes and orders results based on two factors: keywords and geography. Given a query, it identifies pages that contain the keywords *and* whose page footprint intersects with the query footprint, and ranks these results according to a combination of a term-based ranking function and a geographic ranking function that might, e.g., depend on the volume of the intersection between page and query footprint. Page footprints could of course be indexed via standard spatial indexes such as R*-trees, but how can such index structures be integrated into a search engine query processor, which is optimized towards inverted index structures? How should the various structures be laid out on disk for maximal throughput, and how should the data flow during query execution in such a mixed engine? Should we first execute the textual part of the query, or first the spatial part, or choose a different ordering for each query? These are the basic types of problems that we address in this paper.

Before describing our contribution in detail in Subsection 1.5, we first provide some background on web search engines and geographic web search technology. We assume that readers are somewhat familiar with basic spatial data structures and processing, but may have less background about search engines and their inner workings. Our own perspective is more search-engine centric: given a high-performance search engine query processor developed in our group, our goal is to efficiently integrate the types of spatial operations arising in geographic search engines.

## 1.2 Basics of Search Engine Architecture

The basic functions of a crawl-based web search engine can be divided into *crawling*, *data mining*, *index construction*, and *query processing*. During crawling, a set of initial seed pages is fetched from the web, parsed for hyperlinks, and then the pages pointed to by these hyperlinks are fetched and parsed, and so on, until a sufficient number of pages has been acquired. Second, various data mining operations are performed on the acquired data, e.g., detection of web spam and duplicates, link analysis based on Pagerank [7], or mining of word associations. Third, a text index structure is built on the collection to support efficient query processing. Finally, when users issue queries, the top-10 results are retrieved by traversing this index structure and ranking encountered pages according to various measures of relevance.

Search engines typically use a text index structure called an *inverted index*, which allows efficient retrieval of documents containing a particular word (*term*). Such an index consists of many *inverted lists*, where each inverted list $I_w$ contains the IDs of all documents in the collection that contain a particular word $w$, usually sorted by document ID, plus additional information about each occurrence. Given, e.g., a query containing the search terms "apple","orange", and "pear", a search engine traverses the inverted list of each term and uses the information embedded therein, such as the number of search term occurrences and their positions and contexts, to compute a score for each document containing the search terms. We now formally introduce some of these concepts.

**Documents, Terms, and Queries:** We assume a collection $D = \{d_0, d_1, \ldots d_{n-1}\}$ of $n$ web pages that have been crawled and are stored on disk. Let $W = \{w_0, w_1, \ldots, w_{m-1}\}$ be all the different words that occur anywhere in $D$. Typically, almost any text string that appears between separating symbols such as spaces, commas, etc., is treated as a valid word (or *term*). A query $q = \{t_0, t_1, \ldots, t_{d-1}\}$ is a set[1] of words (terms).

**Inverted Index:** An *inverted index* $I$ for the collection consists of a set of inverted lists $I_{w_0}, I_{w_1}, \ldots, I_{w_{m-1}}$ where list $I_w$ contains a *posting* for each occurrence of word $w$. Each posting contains the ID of the document where the word occurs, the position within the document, and possibly some context (in a title, in large or bold font, in an anchor text). The postings in each inverted list are usually sorted by document IDs and laid out sequentially on disk, enabling efficient retrieval and decompression of the list. Thus, Boolean queries can be implemented as unions and intersections of these lists, while phrase searches (e.g., "Big Bang") can be answered by looking at the positions of the words.

**Term-based Ranking:** The most common way to perform ranking is based on comparing the words (terms) contained in the document and in the query. More precisely, documents are modeled as unordered bags of words, and a ranking function assigns a score to each document with respect to the current query, based on the frequency of each query word in the page and in the overall collection, the length of the document, and maybe the context of the occurrence (e.g., higher score if term in title or bold face). Formally, given a query $q = \{t_0, t_1, \ldots t_{d-1}\}$, a *ranking function* $F$ assigns to each document $D$ a score $F(D, q)$. The system then returns the $k$ documents with the highest score. One popular class of ranking functions is the *cosine measure* [44], for example

$$F(D,q) = \sum_{i=0}^{d-1} \frac{\ln(1 + n/f_{t_i}) \cdot 1 + \ln f_{D,t_i}}{\sqrt{|D|}},$$

where $f_{D,t_i}$ and $f_{t_i}$ are the frequency of term $t_i$ in document $D$ and in the entire collection, respectively. Many other functions

[1]This is a slight simplification as the positions of the terms in the queries and documents are often taken into account as part of the ranking function. Our approach is not impacted by this.

have been proposed, and the techniques in this paper are not limited to any particular class. In addition, scores based on link analysis or user feedback are often added into the total score of a document; in most cases this does not affect the overall query execution strategy if these contributions can be precomputed offline and stored in a memory-based table or embedded into the index. For example, the ranking function might become something like $F'(D,q) = pr(D) + F(D,q)$ where $pr(D)$ is a precomputed and suitably normalized Pagerank score of page $D$.

The key point is that the above types of ranking functions can be computed by first scanning the inverted lists associated with the search terms to find the documents in their intersection, and then evaluating the ranking function only on those documents, using the information embedded in the index. Thus, at least in its basic form, query processing with inverted lists can be performed using only a few highly efficient scan operations, without any random lookups.

## 1.3 Basics of Geographic Web Search

We now discuss the additional issues that arise in a geographic web search engine. Most details of the existing commercial systems are proprietary; our discussion here draws from the published descriptions of academic efforts in [30, 32]. The first task, crawling, stays the same if the engine aims to cover the entire web. In our systems we focus on Germany and crawl the `de` domain; in cases where the coverage area does not correspond well to any set of domains, focused crawling strategies [9] may be needed to find the relevant pages.

**Geo Coding:** Additional steps are performed as part of the data mining task in a geographical search engines, in order to extract geographical information from the collection. Recall that the footprint of a page is a potentially noncontiguous area of geographical relevance. For every location in the footprint, an associated integer value expresses the *certainty* with which we believe the page is actually relevant to the location. The process of determining suitable geographic footprints for the pages is called *geo coding* [32].

In [30], geo coding consists of three steps, *geo extraction*, *geo matching*, and *geo propagation*. The first step extracts all elements from a page that indicate a location, such as city names, addresses, landmarks, phone numbers, or company names. The second step maps the extracted elements to actual locations (i.e., coordinates), if necessary resolving any remaining ambiguities, e.g., between cities of the same name. This results in an initial set of footprints for the pages. Note that if a page contains several geographic references, its footprint may consist of several noncontiguous areas, possibly with higher certainty values resulting, say, from a complete address at the top of a page or a town name in the URL than from a single use of a town name somewhere else in the page text. Figure 1.1 shows an example of a page and its footprint.

The third step, *geo propagation*, improves quality and coverage of the initial geo coding by analysis of link structure and site topology. Thus, a page on the same site as many pages relevant to New York City, or with many hyperlinks to or from such pages, is also more likely to be relevant to New York City and should inherit such a footprint (though with lower certainty). In addition, geo coding might exploit external data sources such as `whois` data, yellow pages, or regional web directories.

The result of the data mining phase is a set of footprints for the pages in the collection. In [30], footprints were represented as

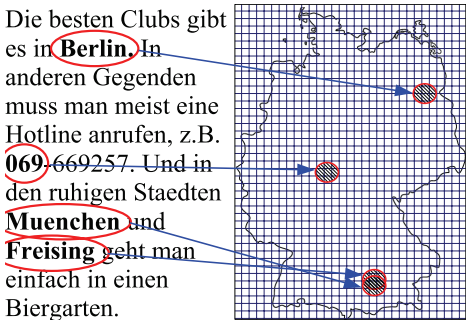


**Figure 1.1: Text inside a web page (left) and the corresponding page footprint (right) for the German web domain.**

bitmaps which were stored in a highly compressed quad-tree structure, but this decision is not really of concern to us here. Other reasonably compact and efficient representations, e.g., as polygons, would also work. All of our algorithms approximate the footprints by sets of bounding rectangles; we only assume the existence of a black-box procedure for computing the precise geographical score between a query footprint and a document footprint. During index construction, additional spatial index structures are created for document footprints as described later.

**Geographic Query Processing:** As in [30], each search query consists of a set of (textual) terms, and a query footprint that specifies the geographical area of interest to the user. We assume a geographic ranking function that assigns a score to each document footprint with respect to the query footprint, and that is zero if the intersection is empty; natural choices are the inner product or the volume of the intersection. Thus, our overall ranking function might be of the form $F''(D,q) = g(f_D, f_q) + pr(D) + F(D,q)$, with a term-based ranking function $F(D,q)$, a global rank $pr(D)$ (e.g., Pagerank), and a geographic score $g(f_D, f_q)$ computed from query footprint $f_q$ and document footprint $f_D$ (with appropriate normalization of the three terms). Our focus in this paper is on how to efficiently compute such ranking functions using a combination of text and spatial index structures.

Note that the query footprint can be supplied by the user in a number of ways. For mobile devices, it seems natural to choose a certain area around the current location of the user as a default footprint. In other cases, a footprint could be determined by analyzing a textual query for geographic terms, or by allowing the user to click on a map. This is an interface issue that is completely orthogonal to our approach.

## 1.4 Query Processing in Standard Engines

Recall that query processing in standard search engines is done by traversing the inverted lists for the query terms, and evaluating the ranking functions on all documents in the intersection of the lists in order to obtain the $k$ highest-scoring results. In large engines the cost of query processing is dominated by the cost of traversing the inverted lists, which grow linearly with the collection size. For example, with about 7.5 million pages per node, the total size of the inverted lists traversed by the *average query* is more than 10 MB per node even after careful compression of the inverted lists [27]. This presents a major performance bottleneck, and a number of techniques have been developed to overcome this.

**Massive Parallelism:** First, all major engines are based on large clusters of hundreds or thousands of servers, and each query is ex-

ecuted in parallel on many machines. In particular, current engines usually employ a *local index organization* where each machine is assigned a subset of the documents and builds its own inverted index on its subset. User queries are received at a front-end machine called *query integrator*, which broadcasts the query to all participating machines. Each machine then returns its local top-$k$ results to the query integrator to determine the overall top-$k$ documents [24]. Note that this results in a very simple and efficient parallelization of the query processing problem, reducing the problem of optimizing overall throughput to that of optimizing throughput locally inside a node, discussed next.

**Caching:** Search engines use caching on several levels. Results for repeated queries may be cached directly at the query integrator, resulting in benefits from 20% to 70% [24, 25, 39, 28, 40]. At each node, frequently accessed inverted lists are cached in main memory [40, 22, 27].

**Index Layout and Compression:** Each inverted list is laid out sequentially on disk or in memory and compressed using one of many techniques that have been proposed [44], though in practice very simple techniques seem to work best [41]. Compression and decompression is typically applied to chunks of some size, ranging from a few KB down to as small as a CPU cache line in CPU-bound scenarios.

**DAAT Query Processing:** When two lists are of similar size, the best way to intersect them is to simply scan both lists. When one list is much shorter, it is better to scan the shorter list and perform lookups into the longer one, assuming elements in the longer list can be accessed individually or in small enough chunks. More precisely, we fetch one element from the shortest list, and then perform a forward seek in the next longer list for a matching docID, and if found another seek in the next list, and so on. This approach, taken by most current engines, is called *document-at-a-time* (DAAT) query processing, and it results in a simultaneous traversal of all lists involved in the query. Whenever an element in the intersection is encountered, we evaluate its score under the ranking function; the top-$k$ results found so far are maintained in a simple heap structure. Note that under this scheme, we may still fetch the entire compressed inverted list from disk unless the forward seeks result in very large skips ahead. However, we save significantly on CPU work by not traversing and uncompressing all retrieved chunks.

**Discussion:** The above optimizations increase query throughput by an order of magnitude, and thus they need to be incorporated into any realistic approach to geographic search query processing. The DAAT approach essentially implements an iterator over documents in the intersection, where all lists are traversed from left to right and the elements in the intersection are produced sorted by docID and without need for intermediate storage. Operators can be ordered in various ways, but are best ordered by selectivity (i.e., shortest list first). It is desirable to preserve this general framework when adding geographic constraints.

One optimization that we do not consider here are *top-$k$ pruning techniques* that attempt to compute (or guess) the correct top-$k$ results without iterating over the entire intersection of the inverted lists, by presorting the lists according to their contributions to the score and terminating the traversal early. There has been a significant amount of work in the IR and database communities on this issue under various scenarios; see [2, 10, 14, 15, 26, 37]. Various schemes are used in current engines, but details are closely guarded. We note that these techniques are highly dependent on a particular choice of ranking function. Pruning techniques for geographic search engines are an interesting topic for future research, but our goal is to first understand the general case.

## 1.5 Contributions of this Paper

We study the problem of efficient query processing in geographic search engines, where each document (web page) consists of a textual part (bag of words) and a page footprint. Formally, a footprint is a function that assigns a nonnegative integer to each location in the underlying geographic domain. A query consists of a set of terms and a query footprint. The query processing problem is to determine the set of documents that contain all the query terms and that also have a nonempty intersection between document footprint and query footprint, and to compute a relevance score according to a given ranking function on *all* such documents.

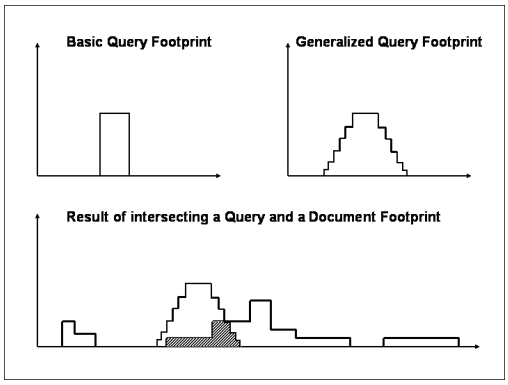


**Figure 1.2: An illustration of footprints in a single spatial dimension. At the top, we have a query footprint with a distance threshold (left), and a footprint for a query that gives a lower score for documents that are farther away (right). At the bottom, we show an intersection computation between a query footprint and a document footprint.**

As discussed in Subsection 1.3, we assume that the ranking function that needs to be evaluated is a monotone combination (usually the sum) of a term-based measure (e.g., cosine or okapi), a global score (e.g., Pagerank), and a geographic score that can be efficiently computed from the exact document and query footprints. Note that we assume `AND` semantics between the term-based and geographic scores, i.e., only documents with non-zero term-based score and non-empty footprint intersection (non-zero geographic score) are considered. While most search engines return only documents containing all textual query terms, our approach does also allow term-based ranking functions based on more general OR semantics. However, all our experiments use AND semantics between terms, and performance might be lower in the OR case. We also require that the precise relevance score is computed for all such documents; thus top-$k$ query approaches based on pruning, which typically compute only approximate scores or precise scores for only a subset of such documents, are not considered. Such approaches are an interesting problem for future research, though we note that the benefits might depend heavily on details of the ranking function that is used.

We argue that the considered setup provides a very flexible approach to geographic search engine query processing. Footprints can be derived in many different ways, and many different geographic ranking schemes can be used as long as they satisfy a few basic conditions. Thus, document footprints can be obtained through

a variety of data extraction, natural language processing, and data mining techniques, with amplitudes scaled appropriately, e.g., based on suitable probabilistic models. On the other hand, different shapes and amplitudes for the query footprint can be used to express different search goals, e.g., any information about events within ten miles of a given location, versus events within say 30 miles but with higher scores for closer results, as illustrated in Figure 1.2. Our main contributions in this paper are as follows:

- We discuss and formally study the query processing problem in geographic web search engines. The problem is also relevant in more traditional information retrieval systems, e.g., when searching a set of news articles or other reports for events relating to a given area.
- We describe several efficient algorithms for query processing in geographic search engines.
- We integrate the algorithms into an existing high-performance query processor for a scalable search engine, and evaluate them on a large web crawl and queries derived from a real query trace. Our results show that even under the fairly general framework adopted in this paper, geographic search queries can be evaluated in a highly efficient manner and in some cases as fast as the corresponding text-only queries.

The query processor that we use and adapt to geographic search queries was built by Xiaohui Long, and earlier versions were used in [26, 27]. It supports variants of all the optimizations described in Subsection 1.4. Since we do do not consider pruning techniques in this paper, this feature was disabled.

In the next section, we describe the data sets that we collected and discuss the resulting challenges in terms of data and memory sizes. Section 3 proposes a sequence of query processing algorithms, and Section 4 reports on our experimental evaluation. Related work is discussed in Section 5, and finally we provide some concluding remarks.

## 2. DATA SETS

We now briefly describe the data sets acquired for our system and our experimental setup, and discuss the resulting performance challenges.

**Documents:** We crawled about 31 million distinct pages from the `de` domain in April 2004, using seed pages from Yahoo's German portal. One issue in German pages are *Umlauts* which `HTML` can represent in multiple ways; these were normalized in a preprocessing step. Apart from the personal background of some of the team members, we chose the `de` domain for two reasons. First, it seemed the right size both geographically and in terms of number of pages. It is quite dense with about 7.8 million registered domains within the relatively small area of Germany. A second reason was the easy availability of geographic databases that can be used for extracting and matching geographic terms.

We retrieved the `whois` entries for all $680,000$ `de` domains in our crawl; many of the 7.8 million registered domains do not actually have a live web server. We also obtained other databases to map telephone area codes, city and village names, and zip codes to coordinates. We used this data to perform geo coding on our document collection, by extracting geographic terms, matching them with the databases, and mapping them to a set of geographic locations and regions (with appropriate amplitudes that express the degree of relevance of the page); more details are given in [30]. Note that the geo extraction and matching steps are at least somewhat language- and country-specific, and benefit from an understanding of language, administrative geography, and conventions for referring to geographic entities.

After geo extraction and matching, about 17 million of the 31 million pages had non-empty footprints based on page content, with an average compressed size of 144 bytes per footprint. After geo propagation across site and link structure more than 28.4 million pages had non-empty footprints.

**Index Construction:** In large search engines, documents are typically partitioned over a number of machines, with each machine receiving between a few million up to maybe 50 million pages. Each machine builds an index on its subset and then performs query processing on this index in a largely independent fashion. We experimented with several different collection sizes; most of the reported results are for a set of about 3.9 million pages on one machine, selected from the 17 million pages with non-empty footprints obtained after geo matching. The resulting inverted index had a size of more than 10 GB total. Thus, the inverted index typically does not fit into memory.

We also assume that in many cases, the complete set of document footprints may not fit into main memory, especially given that a good part of the memory is taken up by other performance-critical tasks such as inverted list caching. However, most footprints can be reasonably approximated by a few bounding rectangles, which consume less than 15 bytes per footprint on average after compression and may often fit into memory. Thus, in our query processing problem, the inverted lists are on disk but efficiently retrievable via scans. Bounding rectangles and spatial data structures based on them are mostly main-memory based (though this is not absolutely needed), but the precise footprints that must eventually be fetched at least for some pages to compute precise scores are on disk under several different data layouts.

**Queries:** Realistic query data is extremely important for meaningful experiments. Since it was impossible to get actual query traces from existing local search engines, we decided to mine an older publicly available AltaVista query log of several million queries for geographical queries. To do this, we ran our geo extraction code on the queries instead of pages, to identify any queries that refer to cities or villages in Germany. The output of this step was then manually cleaned over several days to remove queries that were clearly not geographical in nature (e.g., numerous queries such as "halle berry pictures" were not considered relevant to the German city named *Halle*), resulting in about 10,000 high-quality geographic queries. We then replaced the city names in these queries with footprints of varying diameter and shape around the city coordinates and kept the other search terms in textual form. (We note that with this approach, we are essentially simulating a possible interface for a geographic search engine that analyzes queries to identify queries with geographic meaning.) The resulting set of query footprints is quite different from that obtained by generating footprints at random or according to the footprint distribution of the pages.

An alternative would have been to randomly assign footprints to queries from a query log. A simple example illustrates the problems that would arise form using such synthesized data in our experiments. The Oktoberfest is a famous festival in Munich, and geographic searches for `Oktoberfest` in connection with Munich are thus quite frequent. For the same reason, there are many

web pages about the Oktoberfest in Munich, leading to numerous results for such a query. In synthesized data, a search for `Oktoberfest` might be combined with a random query footprint, say in Berlin, and the query would retrieve far fewer results. Thus, such synthesized queries might traverse a much smaller part of the spatial and textual data structures, making the measured times unrealistic.

# 3. QUERY PROCESSING ALGORITHMS

We now present our query processing algorithms. Recall that a document consists of a set of terms and a document footprint, while a query consists of a set of terms and a query footprint. Our goal is to compute the exact score of every document that (i) contains all query terms and (ii) whose footprint has a non-empty intersection with the query footprint. We assume that the inverted index and document footprints do not fit completely in memory. In order to compute a document's exact score, we thus have to access secondary storage to retrieve its complete footprint and all corresponding entries from inverted lists.

In our algorithms, we model locations in Germany through a $1024 \times 1024$ regular grid, with each tile corresponding to an area of about $700 \times 1000$ meters. Note that this grid model is only applied for the purpose of spatial selectivity during data access. It does not restrict the ability of the ranking function to model smaller distance (e.g., a cafe 50 meters from a location could still be ranked higher than a cafe 200 meters away). That is, the actual footprint resolution can be much finer. Furthermore, footprints can be modeled in many ways, e.g., as bitmaps, as polygons, or using various other approximations. In fact, one of the main advantages of our approach is that it allows an implementation of geographic web search while treating the implementation of footprints and geographic score functions as a black-box.

## 3.1 Naive Algorithms

We now first present two fairly straightforward algorithms that we consider as baseline solutions, discuss their shortcomings, and then later present more optimized solutions.

**Text-First Baseline:** This algorithm first filters results according to textual search terms and thereafter according to geography. Thus, it first accesses the inverted index, as in a standard search engine, retrieving a sorted list of the docIDs (and associated data) of documents that contain all query terms. Next, it retrieves all footprints of these documents. Footprints are arranged on disk sorted by docID, and a reasonable disk access policy is used to fetch them: footprints close to each other are fetched in a single access, while larger gaps between footprints on disk are traversed via a forward seek.

Note that in the context of a DAAT text query processor, the various steps in fact overlap. The inverted index access results in a sorted stream of docIDs for documents that contain all query terms, which is directly fed into the retrieval of document footprints, and precise scores are computed as soon as footprints arrive from disk.

**Geo-First Baseline:** This algorithm uses a spatial data structure to decrease the number of footprints fetched from disk. In particular, footprints are approximated by MBRs, that (together with their corresponding docIDs) are kept in a small (memory-resident) R*-tree. As before, the actual footprints are stored on disk, sorted by docID. The algorithm first accesses the R*-tree to obtain the docIDs of all documents whose footprint is likely to intersect the query footprint. It sorts the docIDs, and then filters them by using the inverted index. Finally, it fetches the remaining footprints from disk, in order to score documents precisely.

## 3.2 Discussion

Experimental results are provided in Section 4, but we can already discuss some potential problems with the naive algorithms at this point and outline possible solutions.

**Non-blocking Spatial Filters:** An advantage of the first algorithm is that both inverted lists and footprints are organized by docID. Data in our DAAT query processor thus can flow between spatial and textual operators without intermediate storage. In the second algorithm, on the other hand, footprint MBRs are organized in a spatial manner. Thus, the inverted index access can only start after *all* output from the R*-tree has been produced. (A similar issue arises if the access order is reversed.) There are however conditions under which the first algorithm performs worse. Assume the first algorithm is confronted with a query consisting of common terms. Without a filter and with footprints organized by docID, it will fetch say $5\%$ of all footprints, a task almost as expensive as sequentially scanning all footprints entirely.

In order to unblock the spatial selection, one can use a simple memory-based table of footprint MBRs sorted by docID, instead of the R*-tree, to filter disk accesses for footprint data. (MBRs for footprints can be stored in much less space than actual footprints.) This results in a non-blocking spatial filter. However, its performance degrades much more drastically than the R*-tree's, if the MBRs grow beyond memory size. As an added benefit, this allows varying the order of operators (textual first, or spatial first, or interleaved) on a query-by-query basis. This is common practice in database systems and DAAT textual query processors, where the most selective filter (rarest keyword in the case of text) is applied first in order to reduce the problem size.

**Toeprints:** There is a significant problem with this simplistic use of footprint MBRs. If a document contains the city names "Berlin" and "Munich", it is bounded by a mostly empty MBR of several hundred miles squared. The existence of many such large MBRs seriously impacts the effectiveness of the spatial structures. For this reason, we will propose to partition each footprint into a small number of disjoint sub-footprints, called *toeprints*,[2] and corresponding MBRs of more limited size; see Figure 3.1 for an example.

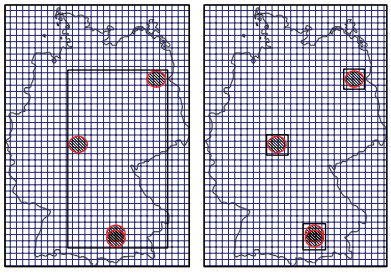

**Figure 3.1: A footprint and its MBR (left) and the resulting toeprints with MBRs (right).**

We tested several different algorithms for splitting footprints into toeprints. Initially, we generated toeprints via a simple recursive partitioning algorithm that stops when (i) no toeprint MBR has a side length larger than some threshold $s_0$, and (ii) no toeprint MBR larger than some threshold $s_1 < s_0$ is more than $x\%$ empty. By

[2] **toeprint** /n./ A footprint of especially small size.

choosing $s_0$, $s_1$, and $x$, we vary size and numbers of the generated toeprints. The resulting toeprints and their MBRs can now be treated as individual objects in our spatial structures and on disk, identified by a toeprint ID. A separate in-memory table can map toeprint IDs to the docIDs of the corresponding pages. During query processing, we first compute separate geographic scores for each toeprint, and later combine the scores of toeprints that belong to the same document.

Note that this imposes a subtle condition on the geographic score function $g(f_D, f_q)$. One must be able to compute the total score by separately computing the score for each toeprint that is intersecting the query footprint. This condition appears to hold for all geographic ranking functions we have considered. (Formally, using the definitions in Subsection 1.3, if footprint $f_D$ is partitioned into disjoint subsets (toeprints) $f_{D,1}$ and $f_{D,2}$, then we require that $g(f_D, f_q) = h(g(f_{D,1}, f_q), g(f_{D,2}, f_q))$ for some function $h()$ with $h(x, 0) = h(0, x) = x$ for all $x$.)

**Data Layout on Disk:** Since we have to compute a precise score for every document in the result set, a large number of document footprints have to be fetched from disk for queries with (i) common search terms and (ii) a query footprint intersecting a densely populated area. For a higher efficiency in these cases, we need a better data layout on disk. Instead of organizing footprints by docID, we store toeprints, and cluster them in a spatial manner. In particular, we propose arranging the toeprint data on disk according to a two-dimensional space-filling curve (see [17, 36] for background). We experimented with two different continuous curves, the so-called Peano and Hilbert orderings. Note that a toeprint can be intersected several times by the curve, since toeprints have a spatial extension. In this case, we assign the toeprint to the part of the curve that takes the longest route through the toeprint. To each toeprint, we assign a toeprint ID by enumerating the toeprints along the curve, and then store all toeprints on disk sorted by this toeprint ID. This layout greatly improves performance by replacing many forward seeks with a few sequential scans. (We also experimented with disk-resident R*-trees but observed much better disk throughput for space-filling curves.)

## 3.3 Improved Algorithms

In the following, we use the above ideas to derive several improved algorithms. All algorithms use toeprints instead of footprints, and toeprints are laid out on disk according to a space-filling curve. We also use simple grid-based spatial data structures in memory; these could be replaced by R*-trees with roughly the same performance.

### *3.3.1 k-Sweep Algorithm*

The main idea of the first improved algorithm is to retrieve all required toeprint data through a fixed number $k$ of contiguous scans from disk. In particular, we build a grid-based spatial structure in memory that contains for each tile in a $1024 \times 1024$ domain a list of $m$ *toeprint ID intervals*. For example, for $m = 2$ a tile $T$ might have two intervals $[3476, 3500]$ and $[23400, 31000]$ that indicate that all toeprints that intersect this tile have toeprint IDs in the ranges $[3476, 3500]$ and $[23400, 31000]$. In the case of a $1024 \times 1024$ grid, including about $50\%$ empty tiles, the entire auxiliary structure can be stored in a few MB. This could be reduced as needed by compressing the data or choosing slightly larger tiles (without changing the resolution of the actual footprint data).

Given a query, the system first fetches the interval information for all tiles intersecting the query footprint, and then computes up to $k \geq m$ larger intervals called *sweeps* that cover the union of the intervals of these tiles. Due to the characteristics of space filling curves, each interval is usually fairly small and intervals of neighboring tiles overlap each other substantially. As a result, the $k$ generated sweeps are much smaller than the total toeprint data. The system next fetches all needed toeprint data from disk, by means of $k$ highly efficient scans. The IDs of the encountered toeprints are then translated into docIDs and sorted. Using the sorted list of docIDs, we then access the inverted index to filter out documents containing the textual query terms. Finally we evaluate the geographic score between the query footprint and the remaining documents and their footprints. The algorithm can be summarized as follows:

**k-Sweep Algorithm:**

(1) Retrieve the toeprint ID intervals of all tiles intersecting the query footprint.
(2) Perform up to $k$ sweeps on disk, to fetch all toeprints in the union of intervals from Step (1).
(3) Sort the docIDs of the toeprints retrieved in Step (2) and access the inverted index to filter these docIDs.
(4) Compute the geo scores for the remaining docIDs using the toeprints retrieved in Step (2).

One limitation of this algorithm is that it fetches the complete data of all toeprints that intersect the query footprint (plus other closeby toeprints), without first filtering by query terms. Note that this is necessary since our simple spatial data structure does not contain the actual docIDs for toeprints intersecting the tile. Storing a list of docIDs in each tile would significantly increase the size of the structure as most docIDs would appear in multiple tiles. Thus, we have to first access the toeprint data on disk to obtain candidate docIDs that can be filtered through the inverted index.

### *3.3.2 Tile Index Algorithm*

This algorithm addresses the limitations of *k-Sweep*. It also uses a simple grid-based spatial structure, but also stores for each tile a complete list of all toeprint IDs that intersect the tile. The resulting increase in the size of the data structure is addressed by using fewer tiles, say $256 \times 256$ or less instead of $1024 \times 1024$, and by compressing the lists in the tiles. More precisely, lists are Golomb-encoded [44]. We also exploit the fact that neighboring tiles have many common toeprint IDs. In addition, we have a table that translates toeprint IDs to their location on disk and to their document ID. (This table could also contain toeprint MBRs, but we found that this has only a very small benefit.) We note that these tables could in fact be stored on disk with fairly small overhead if they do not fit in memory.

This setup allows the new algorithm to first filter docIDs through the inverted index before fetching the toeprint data, and thus decreases the amount of toeprint data fetched from disk. Since toeprints are clustered along the space-filling curve, the decrease in cost will be less than one might expect from the decrease in the number of toeprints that are needed (There is only little benefit in skipping over a few KB on disk). The algorithm can be summarized as follows:

**Tile Index Algorithm:**

(1) Retrieve and sort the docIDs of all toeprints that are listed in any of the tiles intersecting the query footprint.

(2) Access the inverted index to filter the docIDs and corresponding toeprint IDs.

(3) Retrieve the toeprints of all remaining toeprint IDs in an efficient sweep over the toeprint data, using both scans and forward seeks, and compute geographic scores as toeprints are retrieved.

### 3.3.3 Space-Filling Inverted Index

All algorithms so far have treated the inverted index as a black box, and the inverted index lookups essentially result in complete traversals of the inverted lists of the query terms. As we show later, in many cases the total running time of our algorithms is dominated by the textual component of query processing. Thus, to further speed up query processing, we need to reorganize the inverted index itself, and in particular the postings of each inverted list, according to spatial criteria.

One basic insight is that inverted lists are ordered by docID, but that we can assign docIDs to pages in whatever way we choose. (For example, it has been proposed to assign docIDs based on the Pagerank values of the pages, to place pages with high Pagerank at the beginning of the inverted lists [26].) So why not assign docIDs "along a space-filling curve"? In this case, only the geographically relevant chunks of the inverted lists have to be read instead of the entire list.

The simplest approach is to use the toeprint ID, which was assigned based on the space-filling curve, as the docID. If a document has several toeprints, we can simply process the document several times under different docIDs when building the inverted index. As before, we also have a table that maps toeprint IDs to docIDs; this allows us to eliminate duplicates and to combine the geographic scores of toeprints belonging to the same document. Note that this may significantly increase the size of the inverted index, typically by a factor of 2 to 6 depending on the settings, and that this approach might not be appropriate for collections with too many geographic markers per document. The inverted index is typically much larger than the entire footprint data; thus this approach is only a good trade-off if we have enough space and if the textual component of query processing makes up a significant part of the total cost.

This idea can be used in combination with either the *k-Sweep* or the *Tile Index* algorithm. The only change in the algorithm is in the inverted index access, and even this change is "under the hood" of the system: When the toeprint IDs retrieved in Step (1) of *Tile Index* (or Step (2) of *k-Sweep*) are fed into the inverted index in sorted order, the access to the inverted list is much more efficient than before, since most relevant postings are in one or few areas of the inverted list. Our DAAT-style text query processor automatically adapts to this scenario by skipping over large parts of the inverted lists with irrelevant postings.

# 4. EXPERIMENTAL EVALUATION

We now present the result of our experimental evaluation. We ran experiments on several different collection sizes; unless stated otherwise the following results are for a set of about 3.9 million web pages, all of which had non-empty footprints. The resulting size of the inverted index was about 10.1 GB, while the footprints or toeprints had a total size of around 560 to over 700 MB (depending on the toeprint partitioning). We used a Hilbert space-filling curve to lay out toeprint and inverted index data.

Experiments were run on a 3.2 GHz Pentium-4 with a 320 GB Western Digital 3200JD hard disk. Memory was restricted to 512 MB, of which $256MB$ was used for inverted list caching. We ran all results on 100 randomly selected queries, with varying query footprint sizes. We did not allow concurrent execution of multiple queries in our experiments; doing so would produce further slight improvements in query throughput. The numbers reported here also do not include caching of toeprint data in main memory. However, experiments showed an additional moderate reduction in execution time if a small fraction of toeprints are cached.

## 4.1 Comparison of All Algorithms

We start with a high-level overview of the observed average execution time per query, shown in Table 4.1. We see that executing just the textual part of the query using the inverted index takes about 0.33 seconds. Adding geographic query processing in a naive manner, as done in the *Text-First* and *Geo-First* algorithms, results in a large increase in processing time. *Text-First* performs worst, as there is no way to skip any significant parts of the footprint data; the resulting cost is similar to that of simply scanning the entire footprint data (we observed about 50 MB/s speed for a sequential read). *Geo-First* performs slightly better due to its ability to skip part of the footprint data for highly selective queries.

| Algorithm | Time per Query |
|---|---|
| Text Only | 0.33 |
| Text-First | 11.18 |
| Geo-First | 6.47 |
| 1-Sweep | 1.61 |
| 3-Sweep | 0.67 |
| 4-Sweep | 0.61 |
| Tile Index (8x8) | 0.47 |
| Tile Index (16x16) | 0.42 |
| Space-Filling Inverted Index (basic) | 0.35 |
| Space-Filling Inverted Index (improved) | 0.34 |

**Table 4.1: Overview of execution times for various algorithms, on query footprints of size $10 \times 10$.**

The other three algorithms, *k-Sweep*, *Tile Index*, and *Space-Filling Inverted Index*, perform significantly better and achieve processing times approaching that of the simple text query. (We note that our implementation is not yet completely optimized and there is no reason why the time for text-only should be considered a lower bound. Also, concurrent processing of queries would result in additional improvements for both text-only and geo queries.) In the following, we analyze the performance of these three algorithms in more detail and explain how the behavior varies based on different parameter settings.

## 4.2 Detailed Results for k-Sweep

We now first provide some more detailed results for the *k-Sweep Algorithm*. In Figure 4.1 we show results for $k$ equal to 1, 2, 3, and 4. For each $k$, we consider three different settings of $s_0$, $s_1$, and $x$ for generating toeprints from footprints, resulting on average in 4.12, 5.76, and 6.6 toeprints per original footprint, respectively. As we see, performance is much better for 3 and 4 sweeps than for $k = 1$. Performance also increases when we partition footprints into a larger number of smaller toeprints, averaging to slightly more than 0.5 seconds per query under the best setting.

As shown in Table 4.2, this is also reflected in the total amount of data fetched from disk, which is much higher for $k = 1$. We also

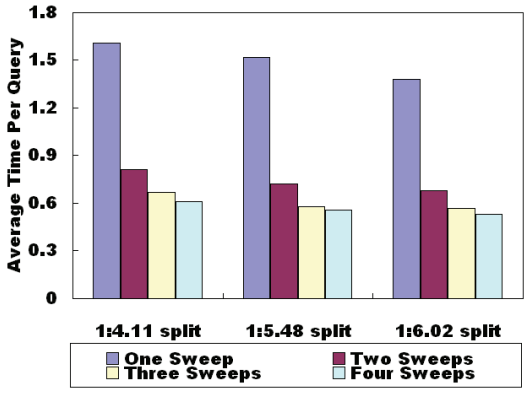


**Figure 4.1: Average query execution time for *k-Sweep* with $k = 1, 2, 3, 4$ and with three different parameter settings for generating toeprints.**

see that the amount of toeprint data read also decreases slightly with smaller toeprints, even though the total size of the toeprint data structures on disk increases slightly.

| Number of Sweeps | 1:1.411 split | 1:1.548 split | 1:6.02 |
|---|---|---|---|
| One Sweep | 61.3 | 58.6 | 51.6 |
| Two Sweeps | 21.6 | 17.0 | 16.2 |
| Three Sweeps | 15.2 | 10.8 | 10.2 |
| Four Sweeps | 12.1 | 9.8 | 8.3 |

**Table 4.2: Megabytes of toeprint data fetched per query.**

As we observed, our simple toeprint partitioning heuristic resulted in a fairly large number of toeprints per footprint on average. To decrease this number, we experimented with several smarter partitioning schemes, in particular one scheme that partitioned toeprints based on the characteristics of the underlying space-filling curve. Thus, this heuristic attempts to generate toeprints that do not intersect several far away parts of the space-filling curve. The results of this improved scheme are shown in Figure 4.2.

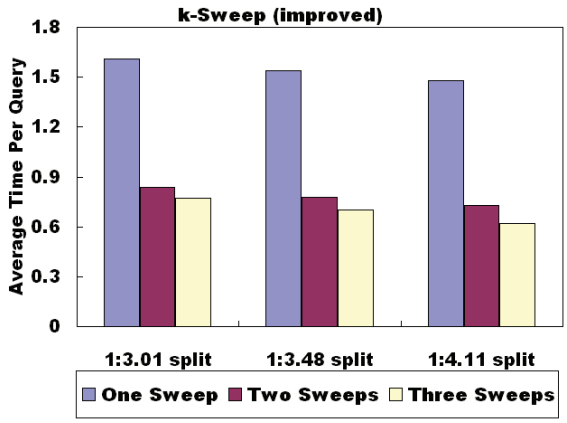


**Figure 4.2: Average query execution time for *k-Sweep* with improved toeprint partitioning heuristic.**

For a fair comparison, these numbers have to be compared to the leftmost group of results in Figure 4.1, which has 4.12 toeprints per footprint. Note however that our real motivation for this better partitioning scheme lies in the other two algorithms, which benefit more significantly as we will see. In addition to Hilbert, we also experimented with another space-filling curve, the Peano Curve, but results are very similar.

## 4.3 Results for Tile Index

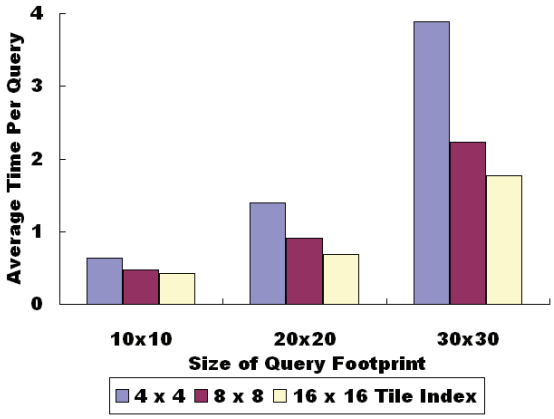


**Figure 4.3: Performance of *Tile Index* for different grid resolutions and query footprint sizes, for a $1 : 6.6$ split ratio.**

Next, we looked at the performance of the *Tile Index* algorithm as we vary the resolution of the grid and the query footprint sizes. As we see in Figure 4.3, using a slightly coarser granularity for the resolution of the tile index results in better execution time. Moreover, it also results in less space for the tile index (not shown in the figure), which is reduced from 196 to about 121 MB uncompressed, or about a third of these sizes in compressed form. As discussed, the tile index does not actually have to reside in main memory, though it is slightly advantageous. The tile index size is also significantly reduced when we use the improved toeprint partitioning heuristic. We also see from Figure 4.3 that query cost increases significantly for larger query footprint sizes since more toeprint data has to be fetched.

## 4.4 Results for Space-Filling Inverted Index

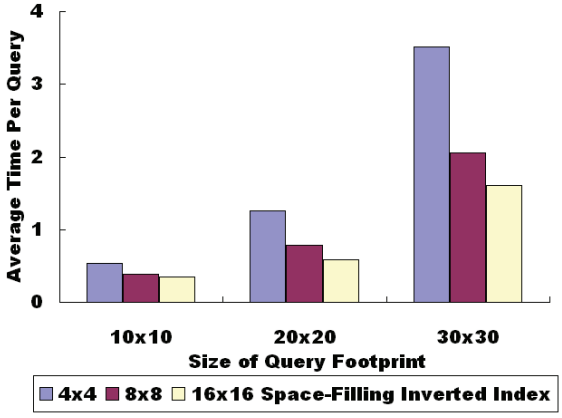


**Figure 4.4: Performance of *Space-Filling Inverted Index* for different grid resolutions and query footprint sizes, with improved partitioning heuristic and $1 : 4.11$ split ratio.**

Next, we looked at the performance of the *Space-Filling Inverted Index* algorithm for different query footprint sizes and different grid resolutions (we implemented the algorithm based on *Tile Index*, but could have also used *k-Sweep* with similar results). We see similar behavior, but overall performance is slightly better. This is also shown by the comparison of *Tile Index* and *Space-Filling Inverted*

*Index* in Figure 4.5, where all algorithms use a $16 \times 16$ grid resolution. We see that use of the improved partitioning heuristic results in better performance particularly for larger query footprint sizes. The reason is that the better heuristic results in a much smaller inverted index: with $1 : 6.02$ ratio, our inverted index balloons to more than 60 GB size, while the $1 : 3.01$ ratio results in an inverted index of "only" a 30 GB. This improved inverted list caching as well as locality of disk accesses. Note that for a $10 \times 10$ query footprint size, performance is almost identical to that of simple text queries, while for larger sizes we are still a factor of 3 to 6 away.

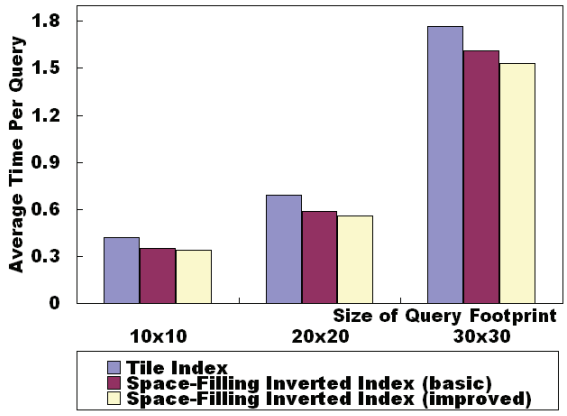


**Figure 4.5: Performance of *Tile Index* and *Space-Filling Inverted Index* with basic and improved partitioning heuristic, for different query footprint sizes.**

# 5. RELATED WORK

We now give an overview of related work in several areas. Many of the references have already been discussed earlier, and thus we keep the description brief.

## 5.1 Search Engines and Query Processing

For background on indexing and query execution in IR and search engines, we refer to [3, 4, 44], and for parallel search architecture to [7, 24, 39]. In particular, [44] provides extensive coverage of indexing, index compression, and basic query execution with term-based ranking functions.

Several recent papers deal with caching in search engines, including list caching [22, 40, 27] and result caching [28, 24, 25, 40, 45]. The performance of inverted list compression schemes is studied in [41, 34, 33, 44]. In particular [41] shows that sloppy but fast schemes tend to outperform more complicated ones in practice. Experimental results in [23] show the performance benefits of DAAT query processing. Recent papers explore top-$k$ ranking schemes based on pruning rather than full traversals of inverted lists; see, e.g., [2, 10, 14, 15, 26, 37].

Recent work in [16] addresses query processing in the case where each page is associated with one or more numeric values, such as price and weight on an e-commerce site, and users specify a range on these values in addition to keywords. The main difference to our work is that [16] assumes only point data, while our footprints have an extension and are often not even contiguous; this significantly complicates the problem. Also, while [16] allows several attributes (dimensions), their algorithm does not use a true multi-dimensional data structure but scans all elements between the lower and upper bound along one of the dimensions. Similar to our work, they extend an existing high-performance DAAT-style text query processor with non-textual constraints while preserving the basic architecture and data flow.

## 5.2 Geographic Web Search Engines

*Google* and *Yahoo* have introduced local search products that appear to focus on retrieval of commercial information, such as stores and restaurants. They seem to make heavy use of yellow page business directories, but exact algorithms are not publicized. Users first retrieve entries for local businesses that satisfy certain keywords, and can then retrieve other pages about these businesses (e.g., restaurant reviews). The Swiss *search.ch* engine [38] is more similar to our approach. The *Geosearch* system [18] is a small academic prototype, with geo coding based on [12]. A more detailed description of the geographic search engine prototype used in this paper appears in [30].

A number of researchers have studied the problem of extracting geographic information from web pages. McCurley [32] introduced the notion of geo coding and describes geographic indicators found in pages, such as zip codes or town names. Subsequent work in [8, 12, 1, 30] proposed additional techniques, including use of hyperlinks and site structure.

An alternative to geo extraction is a Semantic Web approach [13] or the use of HTML meta tags [6, 11]. Our work here does not depend on whether footprints are created through automatic extraction or other approaches, but does assume footprints to be based on coordinates rather than a partition of space into states, counties etc. Note that, as observed, e.g., in [30], the physical locations of web servers are of very limited relevance in geographic search, since in many domains much of the content is served by a few large hosting companies.

## 5.3 Geo Search Query Processing

The definition of geographic query processing in this paper is based on the setup described by us in [30], which also sketched a first very simple geographic query processing algorithm that assumes that all footprint data fits in main memory. Thus, for larger data sets, pages would have to be distributed over several machines, or lossy compression could be applied to decrease the size of the footprints.

Recent work in [42] provides a high level discussion of geographic query processing as part of the SPIRIT project; see also [21] for background. While related, there are several important differences to our work here. First, [42] is focused on retrieving documents in the result set of a query involving textual terms and locations, but does not provide an approach for ranking. Location is modeled on a fairly coarse granularity, with no footprints or other mechanisms for refining and ranking results. Experiments in [42] are performed on a fairly small data set. In contrast, our focus is more on the combination of system and algorithmic issues that arise in a geographic query processor when scaling to very large data sets and high query load.

Another recent paper [31] discusses possible structures for geographic indexing and query processing, including inverted indexes and R$^*$-trees, and proposes future work in this direction, but does not propose any concrete algorithms. Work in [43] discusses a framework for merging different types of rankings, including term-based and (geographic) distance-based rankings, with proposed solutions based on techniques from computational geometry.

Very recently, and concurrent with our work, Zhou et al. [46] have proposed algorithms for geographic query processing under a setup similar to ours. In particular, they also perform ranking using a combination of a term-based and a geographic score, and look at ways to combine inverted indexes with spatial data structures. One major difference is that [46] computes geographic scores based on MBR approximations of page footprints. In our case, the geographic scoring function can be defined in terms of arbitrarily complex "black box" footprint representations with possibly different amplitudes for different coordinates; we use MBRs for indexing purposes only. The footprint representations used in our experiments are on average about $4$ to $5$ times larger per page than the MBRs used in [46], whose experiments show very fast query execution on a data set of about $200,000$ geo-coded pages, compared to almost $4$ million in our case. Thus, the total amount of geographical data is significantly smaller in [46] and could often fit in main memory, while our methods have to fetch more detailed footprints from disk. It would be interesting to see how both approaches compare under comparable experimental setups in terms of efficiency and result quality.

## 5.4 Spatial Index Structures and GIS

There is a vast amount of research on spatial data structures; see, e.g., the survey by Gaede and Günther in [17]. In particular, our algorithms employ spatial data organizations based on R*-tree [5], grid files [35], and space-filling curves - see [17, 36] and the references therein.

A geographic search engine may appear similar to a Geographic Information System (GIS) [20] where documents are objects in space with additional non-spatial attributes (the words they contain). A geographic search could be interpreted as a GIS query: "for this region, find all objects (documents) that fulfill some constraint on the query terms". However, there are significant differences between the performance characteristics of GIS and web search engines that make the former unsuitable for a scalable geographic search engine that answers millions of queries per day on terabyte text collections. GIS are designed for scenarios quite different from geographic web search. There are two straightforward ways a GIS could be adapted for geographic web search. In the first, every term is assigned a layer. All documents that contain the term are indexed as geographic object in the term's layer. Queries are performed by intersecting range queries on the layers for all query terms. The range is given by the query footprint. This approach would follow standard GIS practice, as objects of different nature (e.g., rivers, roads) are usually stored in different layers. However, GIS usually do not support the necessary number of tens or hundreds of thousands of layers needed in our case (the number of somewhat common search terms). The second approach of employing a GIS, retrieving all documents from within the query region and later checking their non-geographic properties, fails as well. In this case, we would store the footprints of all documents as geographic objects in a single layer of the GIS, and then (i) use a range query to retrieve all documents whose footprints intersect the query range, followed by (ii) a filter step that removes those documents that do not contain all query terms. Thus, the GIS query returns unordered document IDs, and textual processing is limited to basic filtering. However, in a geographic web search context, the amount of textual data is at least one order of magnitude larger than the spatial data, making use of an optimized text index structure crucial for performance. Moreover, GIS make assumptions that do not hold for geographic search. They assume space to be relatively sparsely populated with objects, while geo-coded documents can be very dense. Thousands of document footprints may intersect in one location, say, within large cities. Also, objects stored in a GIS only have a single position, while documents may be associated with multiple non-contiguous areas, introducing the problem of duplicates. In summary, we contend that significant changes in GIS system architecture would be needed in order to make them suitable for geographic search.

# 6. CONCLUSION

In this paper we have studied efficient query processing in geographic web search engines. We discussed a general framework for ranking search results based on a combination of textual and spatial criteria, and proposed several algorithms for efficiently executing ranked queries on very large collections. We integrated our algorithms into an existing high-performance search engine query processor and evaluated them on a large data set and realistic geographic queries. Our results show that in many cases geographic query processing can be performed at about the same level of efficiency as text-only queries.

There are a number of open problems that we plan to address. Moderate improvements in performance should be obtainable by further tuning of our implementation. Beyond these optimizations, we plan to study pruning techniques for geographic search engines that can produce top-$k$ results without computing the precise scores of all documents in the result set. Such techniques could combine early termination approaches from search engines with the use of approximate (lossy-compressed) footprint data. Finally, we plan to study parallel geographic query processing on clusters of machines. In this case, it may be preferable to assign documents to participating nodes not at random, as commonly done by standard search engines, but based on an appropriate partitioning of the underlying geographic domain.

## Acknowledgements

We thank Xiaohui Long for access to his high-performance web search query execution software and for help in integrating the new spatial features. We also thank Bernhard Seeger, Thomas Brinkhoff, and Xiaohui Long for earlier collaboration and discussions on geographic search. Yen-Yu Chen and Torsten Suel were supported by NSF CAREER Award NSF CCR-0093400 and by the New York State Center for Advanced Technology in Telecommunications (CATT) at Polytechnic University.

# 7. REFERENCES

[1] E. Amitay, N. Har'El, R. Sivan, and A. Soffer. Web-a-where: geotagging web content. In *Proc. of the 27th Annual SIGIR Conf. on Research and Development in Information Retrieval*, pages 273–280, 2004.

[2] V. Anh, O. Kretser, and A. Moffat. Vector-space ranking with effective early termination. In *Proc. of the 24th Annual SIGIR Conf. on Research and Development in Information Retrieval*, pages 35–42, September 2001.

[3] A. Arasu, J. Cho, H. Garcia-Molina, and S. Raghavan. Searching the web. *ACM Transactions on Internet Technologies*, 1(1), June 2001.

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

[5] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 322–331, May 1990.

[6] D. U. Board. Dublin Core Qualifiers. Recommendation of the DCMI, Dublin Core Metadata Initiative, Jul 2000.

[7] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Proc. of the 7th World Wide Web Conference*, 1998.

[8] O. Buyukkokten, J. Cho, H. Garcia-Molina, L. Gravano, and N. Shivakumar. Exploiting Geographical Location Information of Web Pages. In *2nd Int. Workshop on the Web and Databases (WebDB)*, pages 91–96, 1999.

[9] S. Chakrabarti, M. van den Berg, and B. Dom. Focused crawling: A new approach to topic-specific web resource discovery. In *Proc. of the 8th Int. World Wide Web Conference*, May 1999.

[10] S. Chaudhuri and L. Gravano. Optimizing queries over multimedia repositories. *Data Engineering Bulletin*, 19(4):45–52, 1996.

[11] A. Daviel. Geographic registration of HTML documents, Apr 2001. Internet Draft `http://geotags.com/geo/draft-daviel-html-geo-tag-05.html`.

[12] J. Ding, L. Gravano, and N. Shivakumar. Computing geographical scopes of web resources. In *Proc. of the 26th Conf. on Very Large Data Bases (VLDB)*, pages 545–556, September 2000.

[13] M. Egenhofer. Toward the semantic geospatial web. In *Proc. of the 10th ACM Int. Symp. on Advances in Geographic Information Systems (GIS)*, pages 1–4, Nov 2002.

[14] R. Fagin. Combining fuzzy information from multiple systems. In *Proc. of the ACM Symp. on Principles of Database Systems*, 1996.

[15] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *Proc. of the ACM Symp. on Principles of Database Systems*, 2001.

[16] M. Fontoura, J. Zien, R. Lempel, and R. Qi. Inverted index support for parametric search. Technical Paper RJ10329, IBM, 2004.

[17] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[18] L. Gravano. Geosearch: A geographically-aware search engine, 2003. `http://geosearch.cs.columbia.edu`.

[19] L. Gravano, V. Hatzivassiloglou, and R. Lichtenstein. Categorizing web queries according to geographical locality. In *Proc. of the 12th Conf. on Information and Knowledge Management (CIKM)*, pages 325–333, November 2003.

[20] R. H. Güting. An introduction to spatial database systems. *VLDB Journal*, 3(4):357–399, 1994.

[21] C. B. Jones, A. I. Abdelmoty, D. Finch, G. Fu, and S. Vaid. The spirit spatial search engine: Architecture, ontologies and spatial indexing. In *Proc. 3rd Int. Conf. on Geographic Information Science*, pages 125–139, October 2004.

[22] B. T. Jonsson, M. J. Franklin, and D. Srivastava. Interaction of query evaluation and buffer management for information retrieval. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 118–129, June 1998.

[23] M. Kaszkiel, J. Zobel, and R. Sacks-Davis. Efficient passage ranking for document databases. *ACM Transactions on Information Systems (TOIS)*, 17(4):406–439, October 1999.

[24] R. Lempel and S. Moran. Optimizing result prefetching in web search engines with segmented indices. In *Proc. of the 28th Int. Conf. on Very Large Data Bases (VLDB)*, August 2002.

[25] R. Lempel and S. Moran. Predictive caching and prefetching of query results in search engines. In *Proc. of the 12th Int. World Wide Web Conference*, pages 19–28, 2003.

[26] X. Long and T. Suel. Optimized query execution in large search engines with global page ordering. In *Proc. of the 29th Int. Conf. on Very Large Data Bases (VLDB)*, September 2003.

[27] X. Long and T. Suel. Three-level caching for efficient query processing in large web search engines. In *Proc. of the 14th Int. World Wide Web Conference*, May 2005.

[28] E. Markatos. On caching search engine query results. In *5th Int. Web Caching and Content Delivery Workshop*, May 2000.

[29] A. Markowetz, T. Brinkhoff, and B. Seeger. Exploiting the internet as a geospatial database. In *Workshop on Next Generation Geospatial Information*, October 2003. (Also presented at the *3rd Int. Workshop on Web Dynamics at WWW13*, May 2004.)

[30] A. Markowetz, Y.-Y. Chen, T. Suel, X. Long, and B. Seeger. Design and implementation of a geographic search engine. In *8th Int. Workshop on the Web and Databases (WebDB)*, June 2005.

[31] B. Martins, M. Silva, and L. Andrade. Indexing and ranking in geoir systems. In *Proc. of the 2nd Int. Workshop on Geo-IR (GIR)*, November 2005.

[32] K. McCurley. Geospatial mapping and navigation of the web. In *Proc. of the 10th Int. World Wide Web Conference*, pages 221–229, May 2001.

[33] A. Moffat and J. Zobel. Self-indexing inverted files for fast text retrieval. *ACM Transactions on Information Systems*, pages 349–379, October 1996.

[34] G. Navarro, E. de Moura, M. Neubert, N. Ziviani, and R. Baeza-Yates. Adding compression to block addressing inverted indexes. *Information Retrieval*, pages 49–77, July 2000.

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

[36] C. Ohm, G. Klump, and H. Kriegel. Xz-ordering: A space-filling curve for objects with spatial extension. In *Proc. of the 6th Int. Symp. on Advances in Spatial Databases*, pages 75–90, July 1999.

[37] M. Persin, J. Zobel, and R. Sacks-Davis. Filtered document retrieval with frequency-sorted indexes. *Journal of the American Society for Information Science*, 47(10):749–764, May 1996.

[38] Räber Information Management GmbH. `www.search.ch`.

[39] K. Risvik and R. Michelsen. Search engines and web dynamics. *Computer Networks*, 39:289–302, 2002.

[40] P. Saraiva, E. de Moura, N. Ziviani, W. Meira, R. Fonseca, and B. Ribeiro-Neto. Rank-preserving two-level caching for scalable search engines. In *Proc. of the 24th Annual SIGIR Conf. on Research and Development in Information Retrieval*, pages 51–58, September 2001.

[41] F. Scholer, H. Williams, J. Yiannis, and J. Zobel. Compression of inverted indexes for fast query evaluation. In *Proc. of the 25th Annual SIGIR Conf. on Research and Development in Information Retrieval*, pages 222–229, August 2002.

[42] S. Vaid, C. B. Jones, H. Joho, and M. Sanderson. Spatio-textual indexing for geographical search on the web. In *Proc. of the 9th Int. Symp. on Spatial and Temporal Databases (SSTD)*, 2005.

[43] M. van Kreveld, I. Reinbacher, A. Arampatzis, and R. van Zwol. Distributed ranking methods for geographic information retrieval. In *20th European Workshop on Computational Geometry*, March 2004.

[44] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, second edition, 1999.

[45] Y. Xie and D. O'Hallaron. Locality in search engine queries and its implications for caching. In *IEEE Infocom 2002*, pages 1238–1247, March 2002.

[46] Y. Zhou, X. Xie, C. Wang, Y. Gong, and W. Ma. Hybrid index structures for location-based web search. In *Proc. of the 14th Conf. on Information and Knowledge Management (CIKM)*, pages 155–162, November 2005.