What, Where, and When: Keyword Search with Spatio-Temporal Ranges

Sergey Nepomnyachiy
New York University
sergey.n@nyu.edu

Bluma Gelley
New York University
bgelley@nyu.edu

Wei Jiang
New York University
wj382@nyu.edu

Tehila Minkus
New York University
tehila@nyu.edu

ABSTRACT

With the adoption of timestamps and geotags on Web data, search engines are increasingly being asked questions of "where" and "when" in addition to the classic "what." In the case of Twitter, many tweets are tagged with location information as well as timestamps, creating a demand for query processors that can search both of these dimensions along with text. We propose 3W, a search framework for geo-temporal stamped documents. It exploits the structure of time-stamped data to dramatically shrink the temporal search space and uses a shallow tree based on the spatial distribution of tweets to allow speedy search over the spatial and text dimensions. Our evaluation on 30 million tweets shows that the prototype system outperforms the baseline approach that uses a monolithic index.

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 Performance Experimentation

Keywords

Efficient query processing, Geographic and temporal search engines, Twitter search engines

1. INTRODUCTION

1.1 Motivation

The digital world becomes more mobile every day. As of 2013, 56% of the world's population owns a smartphone, and a full 50% of mobile phone owners use their phone as their primary tool for connecting to the Internet[1]. Twitter reported in late 2013 that 60% of their users access Twitter from a mobile device, and that primarily-mobile users are much more likely than average users to tweet while out of their homes[2]. As more and more web content is generated on the move, it is increasingly tagged with location data as well as creation timestamps. This multi-dimensional data allows for a much richer search experience. Users can query for keywords in documents that were created only within a specific geographic area and period of time. For instance, a presenter at an academic conference may wish to gauge audience sentiment after presenting new research on kangaroos. The researcher wants to know: "What were people tweeting about my paper at this conference?" More formally, the user wishes to retrieve relevant documents (here, we use tweets) for the keywords "kangaroo paper" in the geographical area of Dallas, Texas for the time range between November 3 and November 5, 2014. All the information necessary to answer this query is available in a geo-and-time-stamped document collection, but it is surprisingly difficult to execute on a standard search engine.

Geo-temporally-limited queries are also a golden opportunity for query processing optimization — careful partitioning of the data can allow the search engine to apply early termination techniques and prune large parts of the search space, avoiding costly computations. In this paper, we present 3W, a search framework for geo-temporally tagged data. Our system uses multiple index slices stored in a shallow tree based on the spatial distribution of geo-stamped data to support fast retrieval for location-based queries; it achieves considerable speedups in retrieval for temporal queries by making use of the temporal structure of time-stamped documents. 3W supports low-latency retrieval for queries with spatial, temporal, and textual components. One example of text tagged with both geographic and temporal identifiers is Twitter data. Every tweet has a timestamp, and, since users can opt to have their location (as determined by GPS) attached to their tweets, many have geo-stamps as well. As a social medium, Twitter is primarily used for daily chatter, conversations, and discussions of current events [19]. The spatial and temporal aspects of a tweet, therefore, are essential for providing context to the text. The same tweet can carry different meanings depending on its context: the tweet "eating breakfast" issued from a user's home at 8am

[1]http://www.digitalbuzzblog.com/infographic-2013-mobile-growth-statistics/

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

http://dx.doi.org/10.1145/2675354.2675358

[2]https://blog.twitter.com/2013/new-compete-study-primary-mobile-users-on-twitter

is quite commonplace, but becomes rather unusual when posted from a concert at 9pm. Teevan et al. found that people often search Twitter for "temporally relevant information", including queries that are both highly localized and time-specific, such as real-time local traffic or weather, and information about events that friends were attending [31]. This usage requires a retrieval system designed specifically to efficiently retrieve geo-and-temporally relevant results.

While 3W can be used for any form of geo-temporally annotated documents, we use Twitter data in our evaluations since it vividly illustrates the potential of the system. Unlike traditional web data, where the small amount of spatial and temporal information is dwarfed by an overwhelming amount of text, geo-temporally stamped tweets contain a limited amount of text along with full temporal and spatial information for each document. Rather than serving as additional metadata, the temporal and geographic aspects of a tweet are actually searchable dimensions in their own right.

In addition to its geo-temporal aspects, the Twitter collection also grows at a regular and rapid rate. In our dataset, an average of eight tweets were issued per second in the USA alone. Since Twitter is so dynamic, a Twitter query processor should also have native support for frequent and low-cost updates to incorporate new tweets into the index.

1.2 System overview

The naive (or rather exhaustive) approach to serving geo-temporal range queries with an inverted index is to find all documents matching the keywords (in a conjunctive or a disjunctive query) and then remove from the result set all documents that lie outside of the *cube* defined by the query ranges. In this approach the runtime is proportional to the length of the inverted lists and is not limited by the query ranges. We propose a system that favors *compact* query ranges and provides a significant boost for them. Under our system, query processing time is proportional to the amount of relevant data, and not the entire search space. We compare our system to a baseline implementation using monolithic index and show a significant speedup for compact queries.

Our contribution: [a] A query processing system that supports low-overhead extension of keyword queries with geo-temporal ranges and obtaining significant speedup for *compact* queries. [b] Our system is highly configurable: it can use any off-the-shelf inverted index as a black-box, or only slightly modified. [c] We describe a mechanism for dynamic optimization of data layout for systems where query distributions are volatile.

The paper is structured as follows: In Section 2, we present a brief overview of related work; in Section 3, a high-level view of our system is presented. We evaluate our system in Section 4. Finally, Section 5 discusses avenues for future work, including the possibility of generalizing our approach to other text corpora with geo-temporal data.

2. RELATED WORK

Fast textual search has been studied in depth for around 30 years [1], and various optimization techniques, compression algorithms, and partitioning schemes have been evaluated to boost the efficiency and effectiveness of inverted index based IR systems. Recently, significant research effort has been invested in fast textual search with either temporal or geographic constraints. The former allows users to query for text within a specific time range. A sample query could be "find all documents containing the keyword 'apple' between January and May of 2012". Though some related work has been done towards this motivation in the GIS and information processing communities [23, 26], only a few works on this topic have been produced by the IR community [16, 5]. The latter, geographically constrained search, allows users to constrain text queries within a geographic bounding area. A typical request would be something like "find all documents containing the keyword 'apple' in the city of New York". Among others, [15], [38], [9],[8], [33] and [37] propose solutions for this search problem. Of these papers, [9] is most relevant to our work. We adopt its basic idea of how to use fast textual query processing technique to boost the performance of the geographic search. However, we capture and explore the temporal information as well, proving a significant contribution over existing work.

Across both the information retrieval and database community, only a few [32] [20] [30] [10] [21] have tried to address the problem of searching across both the temporal and geographical dimensions. These suggest indexing both the spatial and temporal dimensions in the context of a static and dynamically-updating environment by using an improved version of a traditional spatial index such as R-trees [17] or KD-trees [2]. However, they do not use the inverted-index based system proposed here.

Busch, et al [5] describe the system design of the Earlybird search engine currently in use at Twitter. This provides some valuable direction in designing and implementing a real-time temporal Twitter search engine, but there is little mention of efficiently exploiting the geographic information embedded in the dataset.

In order to better understand users in Twitter, some studies have focused on Twitter query log analysis, differences between microblog search and web search, and user behavior [31, 19] as well. These papers provided useful guidance as to how to generate a representative query log for evaluation of our system performance.

3. ARCHITECTURE

The naive approach towards geo-temporal search involves first searching a standard textual index of all the data, then filtering the results by the desired geographic and temporal ranges. The problem with this approach is obvious: the computational effort is always a function of the size of the entire dataset. This method completely disregards the potential gains made possible by focusing only on the much smaller portion of the data that actually matches the query's geo-temporal range. We refer to queries with a small geographic and/or temporal range as *compact*.

A common way to achieve these gains is to segment the data based on time, location, or both. An inverted index file is created for each segment, and these files are then arranged in some data structure that allows for easy access to the correct bucket (e.g. a tree [24]). This reduces the search

range somewhat, but it is still dependent on the granularity of the buckets. Designing a system that optimizes the bucket size is not an easy task.

Our system is realized as a variant of an R-tree over the spatial dimension of the data. Each tree node represents a specific geographic area, and all data belonging to that area is contained in or below that node. In particular, we store all the data in leaf nodes, which each contain an independent inverted index for all the documents found in the spatial bounding box defined by that tree node. The tree is kept very shallow to minimize query execution time. In the next section, we describe in detail the construction of the tree and how query processing is performed.

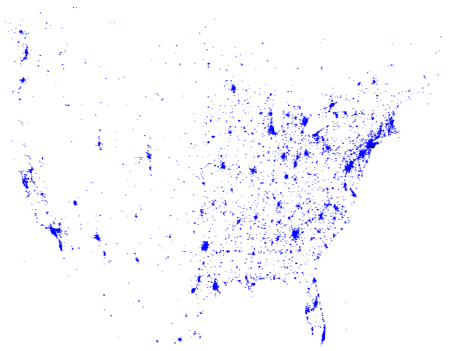

Figure 1: Geographic distribution of tweets

3.1 Spatial Dimension

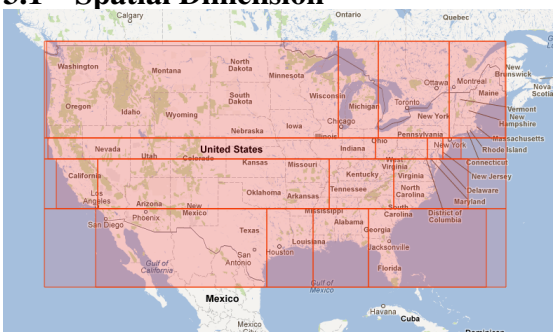

Figure 2: A diagram of the USA spatially partitioned by 3W. Note smaller boxes in areas with high tweet volume

To efficiently manage the spatial component of our data, we use a variant of an R-tree data structure. We split the total area into non-overlapping bounding rectangles of geographical coordinates. As we can see in the example of Twitter data in figure 1 the spatial distribution of user-generated data tends to be highly skewed. This skew means that a grid of equal-area squares is an inefficient way to bucket the data (as suggested as well in [8]). Instead, we split the corpus into bounding boxes with roughly similar *number of documents*, rather than area. The geographic area covered by each bounding box is therefore determined by the number of documents generated in it, and the tree's topology reflects the relative density of documents rather than some more general spatial measure (e.g. square mileage). Our construct resembles (at least in the initial static state) the R*-tree described in [38].

Our tree construction algorithm is described in Algorithm 1. Using a node size threshold T, we construct a non-binary tree. Each child node in the tree represents a geographic bounding box strictly within that of its parent, containing at most T documents. The tree's topology is further governed by a series of tunable parameters which set the number of nodes per level. When T is large enough and these parameters are reasonably set, the tree is shallow and fits easily into cache, making traversal extremely fast. Figure 2 gives a graphical representation of the nested bounding boxes that form our tree.

Algorithm 1 Spatial partitioning of the data

procedure SPLIT
Input:
1. P – set of n points $< x_i, y_i > i \in 1..n$
2. Branching factor at depth i $\{b_0, b_1, ...\}$
3. Threshold T
Result:
balanced tree with $\frac{n}{T}$ leaf-nodes and depth d

#stopping condition – leaf node reached
if $n < T$ **then return**

$d \leftarrow$ *depth of the node*
$S \leftarrow b_d$ subsets of point set P
s.t each subset has $\frac{n}{b_d}$ points **and**
their bounding boxes do not intersect

for all subsets $\in S$ **do**
subtree $\leftarrow$ SPLIT(*subset*) # call split recursively
add subtree as a child of this node

return this node as a root

The tree itself contains no data; each leaf node maintains a pointer to an inverted index for all documents contained within its geographic range. The text index at each leaf node is a standard inverted index structure [1]. For our experiments we use a custom in-house inverted index engine optimized for tweets as described later, however we can integrate any existing off-the-shelf engine. Our system tolerates dynamic updates of the index (especially addition of new documents) if the underlying index engine supports updates. A change in a single document (addition, editing or deletion) in our set-up affects only one inverted index and does not require updating a number of lists proportional to the size of the tree. This is not always possible in systems intertwining inverted index data with the spatial information, where a single update can trigger a multiple updates of different structures (as mentioned in [8]).

3.2 Temporal Dimension

To speed up queries with temporal ranges we ignore any existing document ids and assign new document ids chronologically. Once we have chronologically assigned docIDs, we perform linear regression over all timestamps; this results in a single simple function mapping each timestamp to docID. This trivial conversion step eliminates the need for a specialized structure to store the temporal component. In addition, unlike segmentation of the data, such reordering allows us to perform temporal searches at any level of gran-

ularity. This technique is a form of document reordering, which has been shown to significantly decrease query processing time [35]. When docIDs are assigned in chronological order, all documents in a given temporal range will appear as a contiguous block in an inverted list ordered by docIDs. During query processing, the system will only decompress blocks whose beginning or ending postings are within the desired range. This leads to a significant reduction in query processing time when using a block based compression system or any method that allows the utilization of skiplists. Several techniques have been previously proposed for assigning document identifiers in ways that reduce the compressed size of inverted lists and speed up query processing [12, 28, 3, 4, 27].

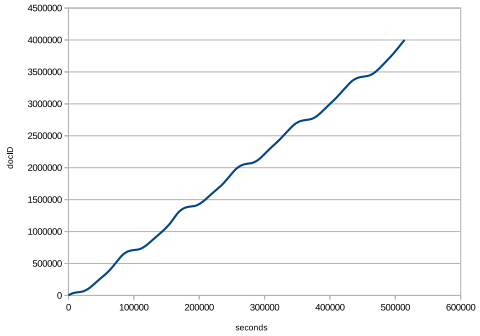

Figure 3: Example of time to docID range mapping for Twitter data

To cope with irregularities in the temporal distribution of documents, and thus timestamps, we introduce *safety margins*. The margins are set to be the largest deviations (positive and negative) from the linear function we learned for our data. When calculating the temporal range for a query, we add these safety margins on each side. In figure 3 we see a slight variation in the number of tweets sent each hour of the day. The safety margins are meant to cope with such local (and repetitive) deviations.

Apart from repetitive small irregularities, however, another cause for inaccurate linear regression (and hence larger margins) is naturally occurring large temporal gaps in the source data. If, for instance, one subset of the documents are from November and the rest are from April, the margins we need to introduce to fix the flaw in the linear regression are extremely large. There are two methods of mitigating this problem: the first is to perform multiple (though still a constant number of) piecewise linear regressions. For example, using 256 regressions for our experimental data reduces the size of the required safety margins by an order of magnitude. This method is helpful when the large margins are a result of either of the above mentioned problems. The second possible way of dealing with large gaps in the document stream is to introduce artificial gaps in the docIDs as well to make the data fit the linear function better. This method is more applicable to the case of a few large gaps in the data.

3.3 Query processing

When processing a query, we traverse the tree from the root, following the path of the nodes whose bounding boxes intersect the query's geo-range. Once the query hits a leaf node, it uses the pointer to the inverted index file of the node to retrieve the inverted lists for all query terms. If the spatial

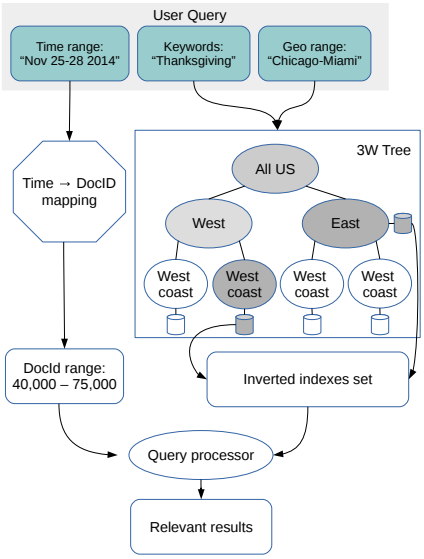

Figure 4: System Architecture overview

range overlaps more than one leaf node, all relevant indexes are queried. The query targets only the range of docIDs that was calculated from the temporal range of the query using the stored regression function (see Section 3.2). Since the leaf nodes queried may not precisely match the coordinates of the user-specified bounding box, the final step is removing all retrieved results outside the chosen geographic area. If the tree is properly optimized based on query volume (see Section 3.5.2), there should not be too many of these.

The query processing operation can be best explained via an example (see Figure 4). Suppose that a user issues the following query: what tweets with the keyword "Thanksgiving" were generated between November 25, 2014 and November 28, 2014 in the area from Chicago to Miami? This query contains three dimensions: geographic, temporal, and textual. The geographic range is the bounding box defined by Chicago as its top left corner and Miami as its bottom right corner. The temporal range is from Nov 25, 2014 to Nov 28, 2014. The keyword is "Thanksgiving". 3W takes the query input and performs the following steps:

Step 1: The temporal component of 3W (explained at length in Section 3.2) exploits the chronological assignment of the docIDs to determine which tweets were issued in the queried time range. In our example, after including safety margins, the final range includes all documents with IDs from 40,000 to 75,000.

Step 2 (in parallel with step 1): The geographic portion of the query searches the spatial tree until it reaches an index-bearing node (or nodes) that intersects its range. Once the range is covered, the inverted indexes from those nodes return all the inverted lists pertaining to the query keywords. In our example, the query traverses past "All USA" down to the nodes which indicate the specific range required; in this case, the nodes "Northeast", "Southeast", and "Midwest" (the number of nodes are reduced in this example for simplicity) are reached and return their respected inverted indexes for the term "Thanksgiving".

Step 3: The query processor takes as input the range of time-appropriate docIDs and the indexes returned by the spatial tree. Utilizing block-skipping methods to efficiently ignore all returned tweets outside the correct temporal range, the query finds all tweets within the range which include the keywords specified. This yields a set of the top-k tweets containing all three components of text, space, and time.

3.4 Text Indexes

Our system is designed to integrate non-intrusively with any inverted index engine regardless of its data layout and compression, scoring system, and query processing algorithms. However, to benefit from all optimizations, the inverted index system must conform with certain expectations. We list them in order of triviality.

1. The inverted index must not intervene with our chronological assignment of docIDs. If the engine assigns its own IDs it will have to provide a constant-overhead mapping to our IDs.
2. The engine should support a mechanism of early termination of the query, given start and end docIDs. In a conjunctive query this feature can be emulated by introducing a dummy inverted list with docIDs from the range.
3. The index engine should provide the option to externalize the lexicon (dictionary) data structure and reduce the query processing initialization times.

The first two are required to support the optimization of search over temporal ranges, while the last one is needed to reduce the space overhead of the supporting data structure of the inverted files compared to a monolithic index.

3.5 Time overhead

3.5.1 Query cost analysis

Query processing consists of two main phases: a trivial phase of finding the nodes of the spatial tree (which is held in memory) which intersect with the query and the second phase of executing keyword search using the inverted index engines. In the simplest case — when a query's spatial range intersects a single leaf node — the query time is inevitably faster than a similar query to a monolithic index system. A query to an inverted index consists of a lookup in a dictionary (a lexicon), assignment of pointers to inverted lists (either pointers to data in memory, or to buffers mapped to files on disk), and the traversal of the lists. In this case, the lookup and assignment require comparable time for a single leaf node and a monolithic index, but the traversal of the index section in a leaf node is much faster than traversing a full index. The speedup is proportional to the ratio of the monolithic index's size and the size of the index in the leaf node.

When a query's spatial range intersects multiple leaf nodes we need to execute a number of keyword searches in the respective index files. Each of these incurs the overhead described above. The worst case is when a user issues a *'whole world'* query hitting every leaf node. In a set-up where all (or most of) inverted files are held in main memory, we do not expect the overhead of multiple query initializations to degrade the running time of a query when compared to a monolithic index. Nevertheless, it is obvious that, conceptually, for such queries a monolithic index outperforms our segmented system. If the files are located on a hard drive the initialization cost can be as high as several milliseconds, hence performing more than a few of them becomes completely infeasible.

3.5.2 Dynamic Restructuring

If we assume that a query stream is likely to include many *spatially]*non-compact queries (that is, with large geographic range) either constantly or fairly frequently, we need to address the issue of high cost for query initialization. As we mentioned, with main-memory-residing indexes the degradation is not significant. In addition, if the lexicon is externalized, as listed in Section 3.4 as a requirement for an index engine, the resulting speedup, along with the shallowness of the tree, are enough to eliminate this overhead.

If the inverted index files reside on disk, the simple solution would be to keep a single monolithic index in addition to the tree and direct all the non-compact queries to it. This static solution is too coarse and effectively doubles the storage footprint for the system. Instead, we propose a dynamic scheme to maintain a data layout that optimizes query runtime with respect to the incoming query stream.

We propose to accumulate statistical data describing the geographic distribution of queries in every node of the spatial tree — namely, how often a query hits all children of a node instead of traversing a single one. At the end of each user defined *epoch*, we traverse the tree hierarchy and perform *merge* or *split* operations on the inverted files. We assume that the visiting patterns in the next epoch will follow a similar distribution to those of the current one, and determine what will be the fastest method of execution. The options are as follows: 1)if we saw many queries hitting most of the leaf nodes in the subtree, it will be faster to query a monolithic sub-index 2)if there were many queries hitting small areas of the bounding box, we would benefit from splitting the relevant leaf nodes to cover even smaller areas, or 3)the system answered the majority of queries efficiently, and the tree structure is therefore optimal. This method allows the system to adapt gracefully to slow changes in the spatial distribution of the query ranges. (Its performance, however, does decrease linearly as the size of the gap(s) increase.)

The same functionality can be used to allow for trading off space vs time optimizations. If some amount of extra space overhead can be tolerated, it is possible to maintain data in the internal nodes as well as in leaf nodes — i.e. after performing a *split* or *merge* operation we keep the source inverted file(s) and can fall back to it later when the distribution changes back. In fact this can be managed as a cache — for a given space budget we can assign priorities to internal nodes and keep their inverted files until the budget needs to be reclaimed by a higher ranking node that evicts the data. In the extreme case this converges to the simple solution — of maintaining a single monolithic index along with the tree data.

4. EXPERIMENTAL EVALUATION

4.1 Twitter-data-specific tweaks

Our Twitter dataset consists of 30 million tweets generated in the continental USA between November 8 and December 1, 2011. The data was collected using Twitter's streaming API. Each data instance contains a unique tweet id, timestamp, latitude and longitude coordinates, and the text of the tweet.

The casual nature of the microblogging platform and its 140-character limit result in text that is very different than standard web data. Abbreviations, elongated words (e.g. "soooooo"), and misspellings abound, many of them deliberate. There have been some attempts to use NLP methods or external lists to normalize tweet vocabulary [18, 34], particularly misspelled words, but we were able to achieve a 60% reduction in index size using the following simple normalization methods: 1) Removing all punctuation and special characters, except @ and #, which carry special significance in Twitter conversation. 2) Discarding all hyperlinks. Since the urls are shortened, each one is unique and does not reflect its destination domain, making it rather useless for search purposes. 3) Compressing all sequences of two or more identical characters to just two. This ensures that multiple variants of misspelled words can be identified as identical (e.g. "loooong" and "looong") without removing the double letters that are part of proper English spelling (e.g "too", "pool", "apple").

As the number of words in a tweet is so small, terms rarely repeat within a document. The frequency data in inverted lists for tweets thus compresses extremely well, as [5] points out. In our dataset, we found that more than 95% of the time, terms occur only once per posting. This allows us to compress only the exceptional 5% whose frequencies are greater than 1. Because most of the inverted lists in our index were short and the gaps between consecutive docIDs were large, we used VarByte [29] to compress the lists. The compression is performed in blocks of 64 postings. We currently support AND queries with results ranked by BM25 [25] scoring. Our framework, however, is not limited to any specific type of search algorithm or index architecture (as long as our docID assignment policy is enforced); in fact, each node could use a different inverted index engine. In the future, we plan to try a standard indexing library such as Lucene. We would also like to introduce the BMW search algorithm [13] to our system to execute conjunctive and disjunctive top-K queries.

4.2 Query Log Generation

The evaluation of 3W was made somewhat difficult by the lack of a publicly available log of microblog queries. As shown by Teevan and colleagues [31], Twitter queries differ substantially from those issued to a standard search engine. Indeed, running a standard TREC query trace on our tweet corpus produced almost no results. The problem is further compounded by the fact that proper testing of 3W requires queries with both temporal and spatial components.

To remedy this, we statistically sampled the dataset and combined the terms to produce a set of artificial queries with a distribution closely mirroring that of the terms in the database. It is unknown whether the Twitter query distribution is related to the term distribution in the underlying set, but we believe it is a logical assumption to make. People are more likely to search for common words than very esoteric ones. In the case of Twitter, this is magnified by the fact that all content is user-created; people tend to search for popular topics, as well as tweet about them [31].

The term distribution in the experimental data has a very long tail. About 50% of terms appear in only a single tweet, and just 0.03% occur in more than 1000 tweets. A closer look shows that most of these low-frequency terms are misspellings or unusual combinations of terms that form an uncommon hashtag. It is unlikely that many people will search for such terms. We therefore focus on the top few percent of terms.

We experimented with generating random combinations of terms. However, because of their random nature, many return no results. This does not accurately reflect a real-world query trace, where many if not most queries do return results; it also artificially improves our speed because traversing non-intersecting lists takes less time. We therefore set out to create a heterogeneous query log that would return at least some results for many or most of the queries. The techniques we used for query log generation are similar to those described in [38, 37].

The likelihood of a conjunctive query returning results decreases rapidly as the number of terms in it increases. In order to create 4-term queries that return results, we restricted ourselves to the single terms with the highest frequencies. Taking the Cartesian product of the top several thousand highest-frequency terms resulted in a massive set of over 7 million two-term queries. We ran all these queries through 3W and recorded the results. Even with queries composed of the product of the top 3% of terms, only about 5% returned a significant number of results. We took the top 3% of queries by number of results and used them to create 3-term queries. We did this by combining each two-term query with a set of terms randomly chosen from a list of the most frequent terms in the dataset. We refer to this process as modified Cartesian product because each multi-term query is combined not with every single-term query but only a set number of randomly sampled terms. This provided us with a very large number of high-frequency queries without the massive overhead incurred by doing a full Cartesian product operation at each step. We repeat the process on the top 5% (by number of results) of the 3-term queries to generate a set of 4-term queries, many of which return results. We filter the queries to remove those with repeated words, then take a weighted sampling from the sets of 1-,2-,3-, and 4-term queries to produce a heterogeneous query trace, varied by number of terms as well as number of results.

4.3 Experiments

3W is designed to optimize search over 3 dimensions without incurring significant space overhead. We compare 3W to two baselines. One, the naive baseline, is optimized over a single dimension, the text. The second, a single-node tree, is designed for optimal performance over two dimensions, text and temporal.

We implemented all the algorithms and the search engines in C++ using BM25 as our ranking function and VarByte

for block-wise compression. We define query processing as returning the top-1000 results of an AND query. The experiments were conducted on a single core of an Intel Xeon server with 2.27Ghz and all data structures reside in memory.

We compare the following systems in our experiments:

Naive baseline: a single inverted index for the entire dataset. All documents that match the keywords are returned and the set is then filtered by the desired space and time ranges. In this case, the engine has to scan the full dataset with each query.

Single-node tree — SN: the result of running our tree initialization algorithm with the splitting threshold (see Section 3.1) set to infinity — the data is never partitioned and is stored in one large index. The difference between this system and the naive baseline is the temporal aspect: docIDs are assigned in chronological order and timestamp-to-docID regression is performed to allow for efficient processing of small temporal ranges.

256 leaf-node tree — 256T: the full 3W system. The tree is constructed by the tree initialization algorithm (Algorithm 1), with splitting threshold T set to 2^{18} and indexes in leaf nodes only. The assignment of docIDs is global and performed in chronological order.

The query log was processed with the naive index and the average query time across multiple tries was 78.3ms.

For the single-node tree and 256 leaf-nodes tree we ran the same query log with 3 different temporal ranges (from short to long: 5 minutes, 1 week, entire time range) and 3 different spatial ranges (from small area to large: NY, 25% of the USA, all of the USA).

	5 mins	**1 week**	**all time**
NY on SN	9.96 (7.9x)	17.86 (4.4x)	67.47 (1.2x)
NY on 256T	0.84 (93.0x)	1.53 (51.2x)	5.90 (13.3x)
25% US on SN	7.81 (10.0x)	17.83 (4.4x)	67.81 (1.2x)
25% US on 256T	1.95 (40.1x)	4.80 (16.3x)	18.24 (4.3x)
all US on SN	6.14 (12.8x)	20.27 (3.9x)	78.34 (1.0x)
all US on 256T	7.78 (10.1x)	24.70 (3.2x)	93.54 (0.8x)

Table 1: Mean runtime of a query (in ms.) with different temporal and spatial ranges for two tested systems. The speedup relative to the naive system is in parentheses

We also created a small handcrafted set of queries and manually evaluated the results returned for them. We found that the results were relevant and usefully ranked.

Single node	*256 leaf tree*	*2x space 256 tree*
32.61	17.61	14.72

Table 2: Mean runtime of a query (in ms.) in mixed log for different systems

In table 2 we show the performance of a system where we allow redundant duplication of the data from leaf-nodes to upper levels of the tree. In this case it is a 2x space overhead scheme. We used a mixed query log with different temporal and spatial ranges allowed.

4.4 Discussion

The mean runtime for the query processing algorithm was 78.3ms, and we observe similar times in the tree experiments of *all time* using a single node tree index. This is expected, as this system does not benefit from spatially compact query ranges and the temporal range in this experiment is the entire time range. For the same reason, there is almost no difference in the performance of a single node tree index across different spatial ranges in 1-week and 5-minutes experiments. We see, however, the effect of the temporal optimization — the speedup of this system relative to the naive baseline is >4x and >7x for 1-week and 5-minutes ranges respectively.

In the *all-time* column of the table for the 256-tree we see the effect of the spatial partitioning scheme, while the 1-week and 5-minutes columns present the runtime of queries compact in both temporal and spatial dimensions. As expected, the smaller the query area, the faster it is processed. We observe speedups of 13.3x, 51.2x, and 93x when using spatial ranges within the NY area for all-time, 1-week, and 5-minutes respectively (when compared with the naive index).

Even for a relatively large area — 25% of the USA — we see significant speedups for the 256-tree: 4.3x,16.3x,40.1x for all-time, 1-week, and 5-minutes respectively.

Note that in the last line we see a degradation in mean query time for the 256-tree index when the entire US range is used. This is the result of the above-mentioned initialization overhead that we must incur when a query is intersecting many (all of them here) spatial nodes. This overhead is more profound in a disk-residing system; in order to be able to observe it in our system (main-memory-residing) we deliberately refrained from optimization of the initialization routine in our implementation of the query processor.

The results in table 2 show how a system with dynamic reconstruction of the tree structure could mitigate this problem. In this experiment we used a system with 2x space overhead allowed. The query log was a mix of all the ranged queries from our first experiment. We see that on average both single-node and 256-tree achieve a speedup relative to the naive index (that runs for 78.3ms), but the redundant tree performs better as it shines in the compact queries (like the non-redundant 256-tree) while avoiding the overhead of queries for large areas (like the single-node tree).

5. CONCLUSION AND FUTURE WORK

Our system efficiently processes multi-dimensional queries over text, space, and time. With a small space overhead and easy integration to an underlying inverted-index-based search engine, it can benefit many systems by early terminating geo-temporal queries. The combined speedup for a system that serves a high volume of queries highly localized in time and space can reach two orders of magnitude when compared to a classical monolithic index. The speedup remains equally impressive even when the underlying inverted

indexes already apply sophisticated early termination techniques. By changing the *branching factor* and *leaf node threshold* parameters, the user can also tweak the system for optimal performance under any kind of different environment, from the most space-starved to the most time-critical.

In the future, we are interested in comparing the performance and compression tradeoffs for using docID reordering locally (on a leaf-node level) rather than globally. The resource-optimization problem of duplicating indexes at higher nodes is also an important area for further investigation. This could allow for more space-efficient automatic tree restructuring. This question is important especially when some of the index files are located on a hard drive. How does query caching affect the speedup we observe in our experiment? A further study with larger query logs can shed some light on this question. We also plan to consider our spatial structure for top-k spatial queries (or nearest neighbor keyword queries) – i.e. where, instead of a query range, a point is given and the engine has to return the k nearest relevant documents. As this kind of query is extremely localized, we believe that our approach can significantly reduce processing time.

Acknowledgements

This work was supported in part by the NSF (under grants 0966187 and 0904246) and by GAANN Grant P200A090157 from the US Department of Education. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of any of the sponsors. The authors also thank Juliana Freire and Dmitri Gromov for their contributions to this paper.

6. REFERENCES

[1] R. Baeza-Yates and B. Ribeiro-Neto, Modern Information Retrieval: The Concepts and Technology Behind Search, 2nd ed., Addison-Wesley Professional, 2011.

[2] J. L. Bentley, Multidimensional binary search trees used for associative searching, Communications of the ACM, vol. 18, no. 9, pp. 509-517, 1975.

[3] R. Blanco and A. Barreiro, Tsp and cluster-based solutions to the reassignment of document identifiers, Information Retrieval, vol. 9, no. 4, pp. 499-517, 2006.

[4] D. Blandford and G. Blelloch, Index compression through document reordering, in Proc of DCC, 2002.

[5] M. Busch, et al, Earlybird: Real-Time Search at Twitter, in Proc of ICDE, 2012.

[6] X. Cao, G. Cong and C. S. Jensen, Retrieving top-k prestige-based relevant spatial web objects, in Proc of VLDB, 2010.

[7] Y. Y. Chen, T. Suel and A. Markowetz, Efficient query processing in geographic web search engines, in Proc of SIGMOD, 2006.

[8] L. Chen, et al. 2013. Spatial keyword query processing: an experimental evaluation. In Proc of VLDB, 2013.

[9] M. Christoforaki, et al, Text vs. space: efficient geo-search query processing, in Proc of CIKM, 2011.

[10] V. Cozza, et al. 2013.Spatio-Temporal Keyword Queries in Social Networks. In collection Advances in Databases and Information Systems. Springer, Berlin. 70-83.

[11] L. R. A. Derczynski, B.Y., and C. S. Jensen. 2013. Towards context-aware search and analysis on social media data. In Proc of EDBT, 2013.

[12] S. Ding, A. Attenberg and T. Suel, Scalable techniques for document identifier assignment in inverted indexes, in Proc of WWW, 2010.

[13] S. Ding and T. Suel, Faster Top-k Document Retrieval Using Block-Max Indexes, in Proc of SIGIR, 2011.

[14] J. Dean, Challenges in Building Large-Scale Information Retrieval Systems (Video), Talk at WSDM 2009 Conference, 2009.

[15] I. D. Felipe, V. Hristidis and N. Rishe, Keyword search on spatial databases, in ICDE, 2008.

[16] F. Gey, et al, NTCIR9-GeoTime Overview - Evaluating Geographic and Temporal Search: Round 2, in Proc of NTCIR-9, 2011.

[17] A. Guttman, R-Trees: A Dynamic Index Structure for Spatial Searching, in Proc of SIGMOD, 1984.

[18] B. Han and T. Baldwin, Lexical Normalisation of Short Text Messages: Makn Sens a #twitter, in ACL 2011, 2011.

[19] A. Java, et al, Why we twitter: understanding microblogging usage and communities, in Proc. of SNA-KDD, 2007.

[20] M. Hadjieleftheriou, G. Kollios and V. J. G. D. Tsotras, Indexing Spatio-temporal Archives, Encyclopedia of GIS, pp. 530-538, 2008.

[21] Lins, Lauro and Klosowski, James T and Scheidegger, Carlos. Nanocubes for real-time exploration of spatiotemporal datasets. In IEEE Transactions on Visualization and Computer Graphics, 19:12, pages 2456-2465, 2013.

[22] X. Long and T. Suel, Three-Level Caching for Efficient Query Processing in Large Web Search Engines, in Proc of WWW, 2005.

[23] I. Mani, et al, Introduction to the special issue on temporal information processing, in TALIP, 2004.

[24] Y. Manolopoulos, A. Nanopoulos and Y. Theodoridis, R-Trees: Theory and Application, Springer, 2006.

[25] C. D. Manning, P. Raghavan and H. Schutze, An Introduction to Information Retrieval, Cambridge University Press, 2009.

[26] Metzler, D, et al, Improving Search Relevance for Implicitly Temporal Queries, in Proc. of SIGIR, 2009.

[27] W. Shieh, et al, Inverted file compression through document identifier reassignment, Information Processing and Management, vol. 39, no. 1, pp. 117-131, 2003.

[28] F. Silvestri, S. Orlando and R. Perego, Assigning Identifiers to Documents to Enhance the Clustering Property of Fulltext Indexes, in Proc of SIGIR, 2004.

[29] F. Scholer, et al, Compression of inverted indexes for fast query evaluation, in Proc of SIGIR, 2002.

[30] Anders Skovsgaard, Darius Sidlauskas, Christian S.Jensen 2014. Scalable top-k spatio-temporal term querying. Data Engineering (ICDE), 2014

[31] J. Teevan, D. Ramage and M. Ringel Morris, #TwitterSearch: a comparison of microblog search and web search, in Proc of WSDM, 2011.

[32] D. Papadias, et al, Indexing Spatio-Temporal Data Warehouses, in Proc of ICDE, 2002.

[33] S. Vaid, C. B. Jones, H. Joho and M. Sanderson. Spatio-Textual Indexing for Geographical Search on the Web. Proceedings of 9th International Symposium on Spatial and Temporal Databases, 2005.

[34] Z. Xue, D. Yin and B. Davidson, Normalizing Microtext, in Proc of AAAI, 2011.

[35] H. Yan, S. Ding and T. Suel, Inverted Index Compression and Query Processing with Optimized Document Reordering, in Proc of WWW, 2009.

[36] J. Zobel and A. Moffat, Inverted Files for Text Search Engines, Computing Surveys, vol. 38, 2006.

[37] D. Zhang, et al. 2013. Scalable top-k spatial keyword search. In Proc of EDBT, 2013.

[38] Y. Zhou, et al. 2005. Hybrid index structures for location-based web search. In Proc of CIKM, 2005.