A Unified Index for Spatio-Temporal Keyword Queries

Tuan-Anh Hoang-Vu
New York University
tuananh@nyu.edu

Huy T. Vo
The City College of New York
hvo@cs.ccny.cuny.edu

Juliana Freire
New York University
juliana.freire@nyu.edu

ABSTRACT

From tweets to urban data sets, there has been an explosion in the volume of textual data that is associated with both temporal and spatial components. Efficiently evaluating queries over these data is challenging. Previous approaches have focused on the spatial aspect. Some used separate indices for space and text, thus incurring the overhead of storing separate indices and joining their results. Others proposed a combined index that either inserts terms into a spatial structure or adds a spatial structure to an inverted index. These benefit queries with highly-selective constraints that match the primary index structure but have limited effectiveness and pruning power otherwise. We propose a new indexing strategy that uniformly handles text, space and time in a single structure, and is thus able to efficiently evaluate queries that combine keywords with spatial and temporal constraints. We present a detailed experimental evaluation using real data sets which shows that not only our index attains substantially lower query processing times, but it can also be constructed in a fraction of the time required by state-of-the-art approaches.

1. INTRODUCTION

The ubiquity of sensors, GPS-enabled smartphones and social networks has led to an explosion in the volume of documents that have both spatial and temporal components. Twitter has over 288 million active users that generate 500 million tweets each day; 80% of these are on mobile devices producing tweets with geographical information [31]. An increasing number of urban data sets are being made available by cities worldwide, and most of these contain spatio-temporal and textual attributes [5]. These open data present new opportunities and can help us better understand different aspects of human life.

Analysis of these data demand complex queries that include constraints specifying regions, keywords, and time intervals of interest. For example, to track how the flu spreads in a city, one can search for tweets that mention flu-related terms in different neighborhoods, over different periods of time [14]. Efficiently evaluating such queries is challenging, in particular, over large volumes of data and when *interactive* response times are required.

Different representations have been used to store and index spatial, temporal and textual information. Text documents are usually modeled as bag-of-words and stored in *flat* structures like inverted indices [25]. Time can be represented in a one-dimensional space. Space, on the other hand, consists of two (or more) dimensions and is often indexed using *space-partitioning* structures such as R-trees [18], grids [27], quadtrees [17], or space-filling curves [34]. A natural approach is thus to use multiple indices to deal with textual, spatial and temporal components, one per attribute type [32]. The results of a query are then produced by taking the intersection of the result sets returned by each index. Besides its simplicity, this approach is also general and supports a wide range of queries. However, this comes at a high cost: many irrelevant documents are likely to be retrieved for queries that contain multiple constraint types. Consider a query such as Q5 in Table 1, which retrieves all tweets containing a term and that was posted at given location and time period. If multiple indices are used, three distinct sub-queries have to be computed separately: all tweets that contain the keyword "ebola" (inverted index), all tweets in New York City (spatial index), and all tweets posted between Sept 1 and Dec 31, 2014 (temporal index). Assuming that a small percentage of tweets in this spatio-temporal slice contain the keyword ebola, a large number of tweets would be retrieved that are not part of the query answer.

To alleviate this problem, techniques have been proposed to combine spatial and textual information in a single index. This class of indices can be broadly classified into two groups: *spatial first*, where a spatial data structure is used to organize the documents over regions and an inverted index is associated with each region [11,16,23,33]; and *textual first*, where documents are first organized by keywords and a spatial structure is used to index the documents associated with each keyword [13,29,34,35]. These strategies were designed to evaluate top-k queries and they are beneficial for queries whose most selective constraints match the primary index structure, i.e., relatively few answers are returned by the primary index, but they can be inefficient otherwise. Consider the queries in Table 1. A spatial-first index speeds up spatial-only queries such as **Q1** and **Q2**, where the spatial constraint is the most selective. For **Q2**, there are many fewer mentions of "ebola" and "outbreak" in New York City than in the rest of the world, therefore filtering over space

CIKM'16 , October 24-28, 2016, Indianapolis, IN, USA
© 2016 ACM. ISBN 978-1-4503-4073-1/16/10. . . $15.00
DOI: http://dx.doi.org/10.1145/2983323.2983751

Table 1: Comparison of different indexing strategies

Query	Description	Constraints	Multi-Index	Spatial first	Textual first	ST2I
Q1	Retrieve *all* keywords mentioned on Twitter within 50 miles of Time Squares	space	✓	✓		✓
Q2	Retrieve the 50 most relevant tweets that contain “ebola” and “outbreak” in New York City	space, text		✓		✓
Q3	Retrieve *all* tweets that contain “ebola”	text	✓		✓	✓
Q4	Retrieve *all* keywords mentioned on tweets posted in Dec 2014 within 50 miles of Times Square	space, time				✓
Q5	Retrieve tweets that contain the term “ebola” posted between Sep 1 and Dec 31, 2014 in New York City	space,time,text				✓

first is advantageous. In contrast, a textual-first approach would be inefficient for this query, since a join would be required between two potentially large result sets: tweets that mention the keywords “ebola” and “outbreak” for the entire world. In contrast, a spatial-first index can be inefficient for queries whose spatial constraints are less selective, i.e., queries that cover regions containing a relatively large number of results. One example is **Q3**, which looks for tweets all over the world; for this query, a textual-first index would perform better. Note that, because these strategies only take text and space into account, they can be inefficient for queries with temporal constraints since an additional index would be required to handle time.

Contributions. In this paper, we present ST2I, a **S**patio-**T**emporal **T**extual **I**ndex structure that supports the *efficient evaluation of both range and top-k queries with multiple constraint types.* It achieves this through a two-pronged strategy. First, by using a single structure to index spatial, temporal and textual attributes together, ST2I is able to uniformly handle different constraint types and filter over multiple dimensions simultaneously, thus, reducing the number of irrelevant documents retrieved and consequently, query execution time. For example, it efficiently evaluates queries such as **Q4** and **Q5** in Table, that contain textual, spatial and temporal constraints. ST2I extends kd-trees in different ways. Unlike traditional kd-tree implementations which are optimized for in-memory access, ST2I was designed to support out-of-core query evaluation. It does so by employing a block-based storage at the leaf node level. This approach retains the flexibility of the kd-tree in supporting multiple dimensions and at the same time scales to large data sets that do not fit in main memory. The memory footprint of the index can be tuned by setting the block size. Another benefit of this structure is that it stores the data (in the blocks) separately from the tree. The tree is thus small and and can fit entirely in memory, speeding up the index look up. As we discuss in Section 3, our implementation of ST2I uses a compact representation for the tree, which further reduces its memory requirements.

Second, to incorporate text into this structure, ST2I uses an efficient technique to map textual information (terms) into numbers. This mapping must be strictly monotone so as to allow the inclusion of the mapped terms into a space-partitioning structure such as the kd-tree. We employ two algorithms to encode and decode the terms that have linear complexity in the size of the terms. The encoding and decoding operations are context free and can be applied on the fly, without requiring intermediate storage or hash tables. In addition, the approach supports evolving collections, where new terms are added dynamically.

We have implemented ST2I and experimentally compared it against state-of-the-art indexing strategies using two real-world data sets: Twitter (8.6GB) and Wikipedia (501MB). ST2I outperforms all other strategies for index construction time, requires a small memory footprint, and makes efficient use of disk space. The experimental results also show that our design decisions for the text mapping and the extensions to the kd-tree structure lead to substantial performance gains. Last, but not least, ST2I leads to very fast query processing times for both range and top-k queries, scaling linearly with respect to the data size and outperforming existing indices by a large margin for queries with multiple constraint types.

Our main contributions can be summarized as follows:

- We propose ST2I, an indexing strategy that efficiently supports range and top-k queries containing textual, spatial and temporal constraints. To the best of our knowledge, ours is the first attempt to efficiently support such queries.
- We introduce a variant of kd-tree structure designed to support out-of-core query evaluation. The index structure is compact and has low overhead for disk accesses.
- We perform an extensive evaluation against state-of-the-art indices using real data sets and report results which show the effectiveness and efficiency of our approach.

The remainder of this paper is organized as follows. Basic definitions are given in Section 2. In Section 3, we present the ST2I index structure, the text encoding method, index construction, and query processing algorithms for both range and top-k queries. We discuss our experimental evaluation and results in Section 4. Related work is reviewed in Section 5 and we conclude in Section 6, where we discuss the limitations of ST2I and outline directions for future work.

2. DEFINITIONS

Given a collection of objects containing spatial, temporal and textual components, and a query q, we aim to efficiently identify the subset of the collection that satisfies q.

Data Model. Let D be a spatio-temporal textual data set. A spatio-temporal-textual (*stt*) object $o \in D$ is represented as a tuple $\langle o.id, o.s, o.t, o.doc\rangle$ where: $o.id$ is the object's unique id, $o.s$ is the spatial component – a *point* in multi-dimensional space; $o.t$ is the temporal component – a *time-point*; and $o.doc$ is the textual content associated with the object, modeled as a bag-of-words $\langle w_1, w_2, ..., w_n\rangle$, where n is the number of distinct words in $o.doc$.

Query Model. We consider two classes of queries: range and top-k queries. Given a *range query* $rq = \langle s, t, text\rangle$, where $rq.s$, $rq.t$, and $rq.text$ are a spatial region, a time interval and a list of keywords, respectively, the result of rq

consists of all objects $o \in D$ satisfying *all* constraints (i.e., $o.s \in rq.s$, $o.t \in rq.t$ and $o.doc$ contains all keywords in $rq.text$). The queries **Q1**, **Q3**, **Q4** in Table 1 are examples of range queries.

Instead of retrieving all items that match the query constraints, a *top-k query* returns only the *k best* results based on a scoring function (e.g., query **Q5**). A top-k query can be represented as $kq = \langle s, t, text, k \rangle$. Similar to a range query it has spatial, temporal and textual components, but it also includes one additional parameter k; kq returns k objects $o \in D$ ranked according to a distance score.

Scoring for Top-K Queries. The *spatial proximity* $ds_s(o, q)$ between an object o and a query q is defined based on the spatial relationship between them:

$$ds_s(o, q) = 1 - \frac{\text{dist}(o.s, q.s)}{\Gamma_S} \quad (1)$$

where $\text{dist}(o.s, q.s)$ is the Euclidean distance between $o.s$ and $q.s$, and Γ_S is the normalization factor, i.e., the maximum Euclidean distance between two points in the dataset D.

The *temporal relevance* $ds_t(o, q)$ between object o and a query q is defined based on their temporal relationship:

$$ds_t(o, q) = 1 - \frac{|o.t - q.t|}{\Gamma_T} \quad (2)$$

where Γ_T is the normalization factor, i.e., the difference between the smallest and largest time points in D.

The *textual relevance* $ds_{text}(o, q)$ between object o and query q is defined based on the query semantics. In this paper, we consider three cases:

- **AND**: If $q.text \subset o.doc$ (i.e., $o.doc$ must contain all keywords in $q.text$), $ds_{text}(o, q) = 1$, otherwise $ds_{text}(o, q) = 0$.
- **OR**: If $q.text \cap o.doc \neq \varnothing$ (i.e., $o.doc$ must contain at least one keyword in $q.text$), $ds_{text}(o, q) = 1$, otherwise $ds_{text}(o, q) = 0$.
- **Distance**: $ds_{text}(o, q)$ is the *distance* between $o.doc$ and $q.text$. Several ranking functions can be employed, such as the well-known cosine similarity or BM25 [25].

The overall distance score between an object o and a query q can then be computed by combining the spatial, temporal, and textual distances:

$$ds(o, q) = \alpha \times ds_s(o, q) + \beta \times ds_t(o, q) + \gamma \times ds_{text}(o, q) \quad (3)$$

where α, β and γ are *normalization factors* that represent the importance of the spatial proximity, temporal and textual relevance, and $\alpha + \beta + \gamma = 1$. These factors are user-defined and input as parameters for each query.

3. INDEXING SPACE, TIME AND TEXT

Different index structures have been proposed to support the efficient evaluation of spatial queries. While these structures can be easily extended to support temporal attributes, the same is not true for textual attributes. Here, we propose the use of a spatial data structure to uniformly handle space, time, and text. We make use of a context-free text mapping algorithm to encode words into a numeric system, or ids, while preserving their alphabetical order (Section 3.1). This allows textual data to be treated as just another numerical dimension of the index, enabling it to be efficiently constructed and queried in a single pass over the data.

Spatial Data Structures. We considered different choices for a spatial index structure, including kd-tree [7], R-tree [18], quadtree [17], and grid index [27]. Quadtree and grid index quickly become inefficient for large data sets that are not uniformly distributed, especially in *high-dimensional* spaces. R-tree-based indices (notably R*-tree [6]) are known for their robustness in the presence of data skew and suitability for disk-based query processing. They are widely used in the spatial extensions provided by database systems. However, they have several drawbacks for high-dimensional spaces. Since each index entry needs to store a minimum bounding rectangle (MBR) for all of its child nodes, the size of MBRs grows linearly as the number of dimensions increases, so does the storage requirement. This translates into a smaller number of indexing entries per block (i.e., the fanout) and reduced efficiency in disk access [36]. The overlapping regions among MBRs also grow rapidly with the increase in the number of dimensions. This can lead to performance degradation, as more false-positive nodes have to be read [9]. While it is possible to optimize R-trees for high-dimensional data, i.e., minimizing overlapping regions while maximizing coverage of MBRs, this problem is non-trivial and computationally expensive. Typically, good splitting strategies [9,18] are quadratic with the number of dimensions, thus making the index construction process less scalable.

A kd-tree [7] is a generalization of a binary search tree used to organize points in k dimensional space. Each non-leaf node splits the points in its sub-trees along a hyper-plane. Similar to binary search tree, points to the left of the defined hyper-plane are present in the left sub-tree and points to the right of the defined hyper-plane are present in the right sub-tree. The canonical way to select the splitting hyper-plane is to cycle through each dimension, and split at the median value of that dimension. This allows kd-trees to support simultaneous filtering over multiple dimensions without the overlapping and complex partitioning imposed by R-trees. For these reasons, we selected a kd-tree-based structure for ST2I. Kd-trees, however, cannot be used to organize data types such as strings. In what follows, we discuss how we addressed this problem.

3.1 Indexing Text

The canonical kd-tree supports indexing for geometric points. In order to index textual data, terms (or keywords) must be encoded as points and the encoding (mapping) method must preserve the alphabetical ordering of the terms. The encoding function f must thus be strictly monotone, i.e., both *bijective* and *monotonic*:

- *Bijective*: For each word w, there exists one and only one associated id $f(w)$ and vice versa. This ensures that we can search every word and that we can map back any index result in the numeric space to the textual space.
- *Monotonic*: For each pair of words w_1 and w_2, if w_1 comes before w_2 in the alphabetical order, then $id(w_1) < id(w_2)$ (e.g., $id(apple) < id(apply)$). Since our indexing structure relies on numeric comparisons to identify matching objects, these comparisons must reflect real textual relations for the index to be meaningful.

A naïve approach would be to extract all distinct terms and sort them before constructing index. Terms can be mapped into their ranks, and a dictionary can be used during query evaluation to translate terms in queries into their equivalent order. However, this requires a full scan of the data

Algorithm 1: word_to_id and id_to_word algorithms

```
// set V to alphanumeric characters
static const char V[] =
  "0123456789abcdefghijklmnopqrstuvwxyz";
// and |V| to the size of V excluding the
// null-terminated character at the end
static const int nV = sizeof(V)-1;
static const int m = 12; // max word length
uint64_t word_to_id(const char *word) {
  uint64_t id=0;
  for (int i=0; word[i] && i<m ; ++i)
    id = id*nV +
         ((word[i]>'9')?
          (word[i]-'a'+10): // letters
          (word[i]-'0'));   // numbers
  return id;
}
void id_to_word(uint64_t id, char *word) {
  word[m] = NULL;
  for (int p=m-1; p>=0 && id; id/=nV, --p)
    word[p] = V[id%nV];
}
```

and sorting a potentially large number of terms. The same expensive process is also required for indexing new terms.

To avoid this inefficiency, we employ two mapping functions for encoding and decoding text that can be applied on the fly. As shown in Algorithm 1, the first algorithm, *word_to_id*, does a single scan over an input string and converts it into a number using the positional notation method. Its complexity is O(m), where m is a constant representing the maximum word length. The second algorithm, *id_to_word*, is responsible for converting IDs back to words and also has O(m) complexity. These algorithms are context-free and can be computed on the fly efficiently without requiring intermediate storage or hash tables.

Implementation Details. To design our encoding mechanism, we took the following observations into account. First of all, words in documents usually have a small number of characters. For instance, the average word length for English language is 5.1 [4]. Similar numbers are found in English literature.[1] The average word length in the Wikipedia and Twitter data sets used in our experiments is 4.9 and 5.6, respectively. Besides, the number of words having more than a certain number of characters follow a power law distribution. As Figure 1 shows, shorter words are more popular than longer ones. Secondly, all English words can be expressed by alphanumerics and a limited set of special symbols (e.g., hashtag (#) and hyphen (-)). Note that this is only a small subset of the ASCII standard, and can be encoded using fewer bits than the usual single-byte representation.

The value of m (12) was carefully chosen so that the corresponding number for any word of length 12 or less containing alphanumeric characters can be stored using native 64-bit numbers. Figure 1 shows that less than one percent of words in Twitter and Wikipedia have more than 12 characters. Therefore we choose to encode the first 12 characters of any word instead of all characters. For example *international* and *internationalization* are encoded using the same number. At query

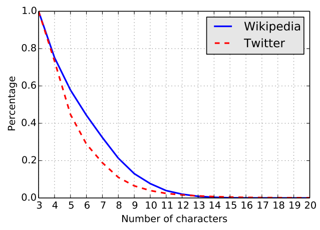


Figure 1: Power law distribution of word length.

[1]http://languagelog.ldc.upenn.edu/nll/?p=3532

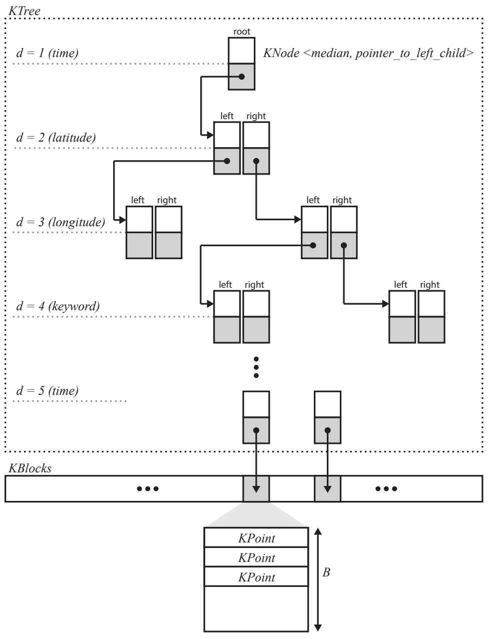


Figure 2: Structure of ST2I.

evaluation time, additional word comparisons are required to return the correct results. As we discuss in Section 4.3, the overhead for these comparisons is negligible. Note that the mapping functions work for English and for languages that use Latin characters, but they are not suitable for languages that contain non-latin characters.

3.2 The ST2I Index

The structure of ST2I is illustrated in Figure 2. It consists of two parts: the tree (**KTree**) and the data (**KBlocks**). KTree is similar to traditional kd-trees, but our implementation uses a more compact representation. There are two types of nodes (**KNodes**): internal nodes and leaf nodes. Each internal node stores the splitting point of the current dimension and a pointer to the left child node. As we describe below, we store a single pointer for both left and right child nodes. Each leaf node stores a pointer to a KBlock. Even though internal nodes and leaf nodes have different semantics, they are stored using the same data structure (leaf nodes have splitting point value set to $NULL$). Each KBlock is a fixed-length array of size B and stores a list of **KPoint** objects. The KPoint for an object $o \in D$ is denoted as $\langle o.id, o.s, o.t, word\_to\_id(w), meta\rangle$, where w is a term in $o.doc$. A KPoint consists of two parts. The first part is a point in n-dimensional space, where $n = k + 2$, k is the number of dimensions for the spatial component, and the two additional dimensions correspond to the temporal and textual components. The second part, which we call *meta*, can be used to store additional information that can vary for different data and query types (e.g., *word frequency* or *tf-idf* in top-k queries). A KPoint is created for each term in *o.doc*. While this leads to some redundancy – the space and time components in o are repeated in all KPoints corresponding to o, it enables ST2I to uniformly handle different query constraints and filter over multiple dimensions simultaneously.

In Figure 2, the first level of KTree splits the data set by *time*, i.e., the left part of the tree contains KPoints having their time value less than the median, and the right part

Algorithm 2: st2i_build

```
1   Input: $P, root, d$
2
3   if $|P| \leq B$
4       $root.content \leftarrow$ new Block($P$)
5       return
6   $cur\_d \leftarrow d \mod n$ % \label{tc:mod}
7   $tmp \leftarrow$ get_value($P, cur\_d$)
8   $median \leftarrow$ nth_element($tmp, |P|/2$)
9   $left \leftarrow 0$
10  $right \leftarrow (|P| - 1)$
11  while $left < right$
12      while get_value($P[left], cur\_d$) $\leq median$
13          $left \leftarrow left + 1$
14      while get_value($P[right], cur\_d$) $> median$
15          $right \leftarrow right - 1$
16      if $left < right$
17          swap($P[left], P[right]$)
18  $root.median = median$
19  **st2i_build**($P[0...right], root.left, d + 1$)
20  **st2i_build**($P[right + 1...|P| - 1], root.left + 1, d + 1$)
21
22  return
```

Algorithm 3: st2i_search

```
1   Input: $q, root$
2   $C \leftarrow \{\}$
3
4   if $q.t = \varnothing$
5       $C \leftarrow C \cap$ **st2i_traverse**($q, root, 0$)
6   else
7       for each $w \in q.keywords$
8           $id_w \leftarrow$ **word_to_id**($w$)
9           $q_w \leftarrow \langle q.p, id_w \rangle$
10          $C \leftarrow C \cap$ **st2i_traverse**($q_w, root, 0$)
11
12  return **st2i_process**($C$)
```

of the tree contains KPoints having a value greater than or equal the median. Recursively, the data set is split on the second level by *latitude*, third level by *longitude*, fourth level by *keyword* (using word_to_id(keyword)), and on the fifth level by *time* again. The process continues until the number of data points being considered is less than the KBlock size B. In what follows, we discuss the advantages of ST2I compared to other kd-tree variants [24, 26].

Separation of the Tree Structure and Data. Together with using a compact representation (see below), separating the tree structure and data leads to a substantial reduction in the size of the KTree. A KTree is usually orders of magnitude smaller than the actual data, and thus, it can fit into memory, allowing efficient traversal of the index. To derive the answers to a query q, only KBlocks with data relevant to q are accessed.

Compactness of Node Data. ST2I only requires 2 values (integers) per node, compared to 4 values used in [36], and 3 values used in popular libraries like *nanoflann* [2] and *Spatial C++ Library* [3]. Since we allocate space for the two children of each node in a pair, we only need to store a single pointer for both of them. This makes the index more compact, leading to *higher memory locality* and *faster traversal time*. As a point of reference, storing 2 values instead of 3 values saves over 2GB for a 100 million tweets data set (see Section 4).

Flat Tree Data Structure. We use a single breadth-first ordered array to store the KTree nodes. This is similar to the storage method used for binary heaps. The root is stored at position 0. A node stored at position i will have its left child (if exists) stored at position $2 * i + 1$ and its right child at $2 * i + 2$. The array can be simply read sequentially as deeper levels of KTree are traversed. As we describe below, our KTree construction algorithm stores KNodes in the same order as the traversal algorithm accesses them. Combined with the fact that non-leaf and leaf nodes use the same data structure, this method results in a compact storage with high-locality of access and low disk overhead.

Use of Memory-Mapped Files. Since ST2I uses a flat tree data structure and node size is fixed, the approximate location of any node can be calculated efficiently. To support out-of-core queries seamlessly, in our implementation, we make use of 64-bit memory-mapped files that are available in current operating systems. By using kernel-space mapped files, we avoid making copies of data in user-space and leave the memory management to the OS.

Complexity. The KTree is balanced: each level of the KTree is split at the median value of each splitting plane, thus the available data points are partitioned equally between the left and right child. Recall that the ST2I uses a block-based storage [26, 28] at the leaf nodes. This is in contrast to the original kd-tree design, where each leaf node links to a single record. This modification allows our data structure to align better with external memory models. In addition, we also use an implicit strategy of dimension interleaving to decide which axis to be split at each level. The benefits are twofold: this improves worst-case complexity for skewed data sets and reduces the storage cost of the KNodes. Let N be the number of data points and B be the maximum number of points per leaf, the maximum depth and the maximum number of nodes in the KTree are $k \log(\frac{N}{B})$ and $\frac{N}{B}$, respectively.

In summary, the benefits of ST2I are: (1) since data points are stored in blocks, it is I/O friendly with a small footprint; (2) it is cache friendly, because nodes are stored in a breadth-first order array relevant KPoints are likely to be loaded together; and (3) the index structure can be accessed quickly through memory-mapped files leading to increased performance and seamless support for disk-based storage.

3.3 Index Construction

Index construction is accomplished by two algorithms: *Point Creation* and *Tree Construction*. Point Creation makes a *single* scan through data set D and converts each spatio-temporal textual object o into a list P of KPoints. Unlike other approaches, this algorithm requires no prior knowledge of (or statistics about) the data set. The second algorithm, Tree Construction, uses the list P of KPoints as input and constructs the KTree. Tree Construction is similar to the canonical k-d tree construction, but since we use a linear selection algorithm [10] to find the median value for each splitting plane, it requires much less time to process each dimension ($O(N)$ instead of $O(N \log N)$).

The Tree Construction, **st2i_build** (Algorithm 2) receives as inputs the Point array P, the *root* node and the depth d of the subtree it is building. It uses two additional parameters: n, the number of dimensions ($n = k + 2$); and B, the KBlock size. The algorithm iterates over the dimensions in the n-dimensional space. In lines 7 and 8, the keys corresponding to the current dimension are extracted and their *median* value computed. Lines 9 to 17 perform in-place swappings on P to partition all points around the pivoting median point. Lines 19 to 20 recursively construct the left and right subtrees of the next dimension.

Algorithm 4: st2i_traverse

```
Input: $q, root, d$

if $root.content \neq \varnothing$
    return $root.content$
$C \leftarrow \{\}$
$cur\_d \leftarrow d \mod n$ % \label{tc:mod}
if $(-\infty, root.median] \cap$ get_value$(q, cur\_d) \neq \varnothing$
    $C \leftarrow C \cap$ **st2i_traverse**$(q, root.left, d+1)$
if $(root.median, \infty) \cap$ get_value$(q, cur\_d) \neq \varnothing$
    $C \leftarrow C \cap$ **st2i_traverse**$(q, root.left+1, d+1)$

return $C$
```

Top-k Query Support. Different from range queries, to process top-k queries the algorithm needs to rank each candidate based on the scoring function. Since the textual relevance function requires word frequency, we make use of the *meta* field of KPoint to store relative frequency of each word in each object's document.

Complexity. **st2i_build** works in a divide-and-conquer fashion similar to quicksort, but it always performs balanced partitioning. For each partition, the complexity is linear, bounded by the selection algorithm and the swapping process. Let $M = \frac{N}{B}$ be the number of leaf nodes in our k-d tree, the amortized cost of building the index structure at each dimension would be exactly $O(M)$. Since there is approximately $k\log(M)$ levels, the total complexity of **st2i_build** is $O(kM\log(M))$.

3.4 Query Processing

Since the spatial, temporal, and textual components of each object are integrated in the ST2I structure, the query processing algorithm can map any spatio-temporal keyword query into a range search query regardless of search criteria. To reduce the query processing time, the algorithm reads and navigates the tree in the same order as it stored. For each query, ST2I starts from the root of KTree and moves down recursively, similar to the process used for index construction. At each level of the KTree, the algorithm goes left or right depending on the split value of the current node and the value of the query constraint in the current dimension. When a leaf-node is reached, all KPoints in the KBlock linked from it are added to the candidate list. After the algorithm finishes traversing the KTree, each KPoint in the candidate list is evaluated to check whether it satisfies all query constraints. The algorithm then outputs the valid results. Evaluation of range queries and top-k queries are described below. For top-k queries, early termination and candidate ranking strategies are employed to yield better overall performance.

Range Query Processing. Single-term and multiple-term queries are evaluated in a similar fashion. For each keyword w in $q.keywords$, **st2i_search** (Algorithm 3) converts it into id_w using the **word_to_id** algorithm and creates an individual range search query q_w from $q.s$, $q.t$ and id_w. Then, for each q_w, **st2i_search** traverses KTree, generates the candidate KPoints (using Algorithm 4 **st2i_traverse**), and adds them to the global candidate set C. Finally, depending on the textual relevance function being used, **st2i_search** either intersects, unions or calculates the similarity score for candidates from different keywords, and outputs the final results.

Algorithm 4 details the **st2i_traverse** procedure. In addition to the search query q and the current node $root$, **st2i_traverse** also keeps track of the current depth d. Lines 3 and 4 check if a leaf-node has been reached. If so, all points in the linked block are returned as candidates. Otherwise, lines 7 to 10 compare the value of the current node (i.e., $root.median$) with the value for the query constraint in the split dimension cur_d, and recursively traverse left, right or both, until a leaf-node is reached.

Algorithm 5: topk_st2i_search

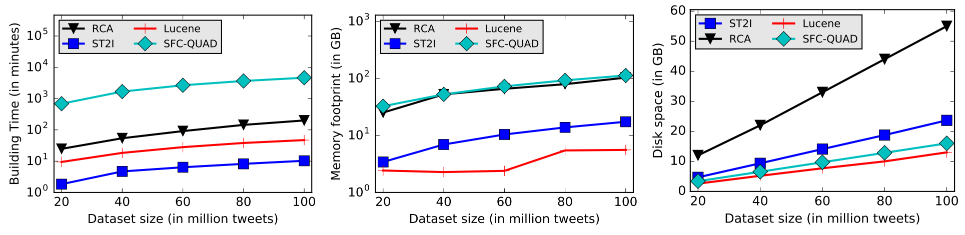
```
1   Input: $q, root$
2   $outer\_upperbound \leftarrow 1$
3   $topk\_lowerbound \leftarrow 0$
4   $cur\_layer \leftarrow 1$
5   $TOPK \leftarrow \{\}$
6
7   while $topk\_lowerbound < outer\_upperbound$
8       $cur\_radius \leftarrow cur\_layer \times STEP$
9       update radius of $q$ based on $cur\_radius$
10      for each $w \in q.t$
11          $id_w \leftarrow$ **word_to_id**$(w)$
12          $q_w \leftarrow \langle q.p, id_w \rangle$
13          $C \leftarrow C \cap$ **st2i_traverse**$(q_w, root, 0)$
14      for each $c \in C$
15          compute combined score of $c$
16          append $c$ to $TOPK$
17      update $outer\_upperbound$
18      update $topk\_lowerbound$ of $TOPK$
19      $cur\_layer \leftarrow cur\_layer + 1$
20
21  return $TOPK$
```

Top-k Query Processing. The key idea behind the optimization techniques for spatial top-k queries is to partition the search space into layers relative to the position specified in the search query. Then, the algorithm only retrieves and validates candidates one layer at a time, thus avoiding computation and validation for answers not among the top-k. For each layer, we need to compute the lower bound of the top-k candidate list and the upper bound of all other points outside of the layer. The algorithm stops when the lower bound from the top-k is larger than the upper bound of the remaining points.

In Algorithm 5, we present a variant of **st2i_search** called **topk_st2i_search** to support top-k queries. $TOPK$ is a priority queue that stores the current best $q.k$ objects as well as the upper bounds and lower bounds of its members. In each iteration, **topk_st2i_search** expands the current search radius by $STEP$ – the unit size of the divided spatial layers. Assuming that the spatial search space is divided into η layers (i.e., it takes η iterations to go over all points in the search space), $STEP = \frac{\Gamma_S}{\eta}$. Then for each keyword in the query, **topk_st2i_search** traverses through the KTree and yields candidate list. The score for each candidate c is calculated by a scoring function and the pair $\langle c, score(c)\rangle$ is added to $TOPK$. At the end of each iteration, the lower bound of $TOPK$ and the upper bound of the outer layer are updated. The upper bound of outer layer is calculated based on the upper bound of the spatial proximity, temporal relevance and textual relevance as follows:

$$outer\_upperbound = \alpha \times (1 - \frac{cur\_radius}{\Gamma_S}) + \beta \times 1 + \gamma \times 1 \quad (4)$$

where $\alpha \times (1 - \frac{cur\_radius}{\Gamma_S})$ is the maximum score of any objects in the next layer; $\beta \times 1$ and $\gamma \times 1$ are the maximum possible temporal relevance and textual relevance, respectively. The algorithm terminates when the maximum possible score of any object in the outer layer is smaller than the minimum score of all elements in $TOPK$.



(a) Building time (log scale) (b) Memory footprint (log scale) (c) Disk space

Figure 3: Index construction performance for Twitter data sets.

4. EXPERIMENTAL EVALUATION

To assess the efficiency of ST2I, we compare its performance against state-of-the art techniques using real-world data sets collected from Twitter and Wikipedia. We consider different aspects, including index construction, memory footprint, disk space and query processing time. We also study the effectiveness of our design decisions, in particular, the block-based structure and text-encoding mechanism.

4.1 Experimental Setup

State-of-the Art Approaches. We compared ST2I to different strategies for both range and top-k queries: *RCA* [34], a textual-first index that uses space filling curves for spatial components; *SFC-QUAD* [13], a spatial-first index that uses quadtree for spatial components and space filling curves for textual components; and *Lucene 4.10.3* [1], an open-source index that uses separate data structures for spatial, temporal and textual components. RCA was designed to process top-k queries and were shown to be the best performing strategy [34]. SFC-QUAD was shown to be the best performing strategy for range queries [12].

Not all approaches are optimized for all types of queries or support all constraint types. For example both RCA and SFC-QUAD do not support temporal constraints. RCA is optimized for top-k queries and SFC-QUAD is optimized for range queries. Therefore, to ensure a fair comparison, we tested each approach only with queries they support and optimize. We evaluated the SFC-QUAD implementation by Chen et al. [12]. Implementations for the other approaches were obtained directly from their authors.

Hardware and Software Configuration. We conducted experiments on a PC with dual Intel Xeon E5-2695 2.4GHz processors–12 cores in total, 16 TB of disk space and 128 GB of main memory. The OS used was Fedora 19 kernel version 3.14.19-100. ST2I was implemented in C++ and was compiled using GCC 4.4.6. Other approaches were compiled using Oracle Java 1.7.0_25. We placed no CPU, memory or I/O restrictions.

Data Sets. We experimented with two data sets. The Wikipedia data set contains 538,176 Wikipedia articles with geo-location extracted from Wikipedia database dumps on August 11, 2014.[2] The Twitter data set contains 100 million geo-tagged tweets collected using the Twitter Public Stream API[3] over the course of 2 months (April - May, 2014). Details about the data sets are given in Table 2.

Queries. Since there are no publicly available workloads for spatial-temporal-keyword queries, we generated our own. First, keywords and spatial locations with different levels of popularity were selected and put into separate candidate pools. We then created different query workloads that correspond to different combinations of spatial and textual popularity levels: *EASY* (i.e., unpopular locations and unpopular keywords) and *HARD* (i.e., popular locations and popular keywords). Query workloads that have other combinations of locations and keywords yield similar results as *EASY* and *HARD* workloads and have been omitted. This workload allows us to study the behavior of the indexing strategies for a wide range of queries with varying selectivity for the different dimensions. Additional parameters were generated for range and top-k queries. We executed each query workload and took the average query time for each index. Querying times are measured in milliseconds and do not include index loading time.

[2]https://dumps.wikimedia.org/enwiki/20140811
[3]https://dev.twitter.com/streaming/public

Table 2: Properties of data sets used in evaluation.

Data Set	Objects	Distinct words	Distinct words per object	Size
Twitter	100,000,000	6,774,156	6.54	8.6 GB
Wikipedia	538,176	1,175,293	128.96	501 MB

4.2 Index Construction

For each approach, we measured the index construction time, memory footprint and disk space usage. The results for the Wikipedia data set are summarized in Table 3. For the Twitter data set, we also studied the scalability of the approaches by varying the data size from 20 to 100 million tweets. The results are shown in Figure 3.

Construction Time. Figure 3a shows the time (in log scale) each strategy needed to build indices for different data sizes. The time is measured in minutes and includes the time for loading data, building index, and writing index to disk. ST2I outperforms all other strategies. The scalability obtained by ST2I can be attributed to the context-free text mapping algorithm and the unified kd-tree structure. For instance, ST2I took 10 minutes to build the index for 100 million tweets – 4 times faster than Lucene, which took 47 minutes. Lucene needs to build separate structures for spatial, temporal and textual components. RCA, which is a textual-first approach, creates a spatial structure for every keyword, thus requiring massive amount of memory and a large number of disk access. SFC-QUAD does not scale well as the data increase. Its construction algorithm spends substantial time arranging documents into Z-order, before building a standard block-compressed inverted index; this compression also slows down the construction process.

Memory Footprint. Figure 3b shows the memory footprint (in GB) for each index during construction. We periodically measured the memory footprint and selected the peak value. Lucene has the smallest footprint, as a result of

Table 3: Index construction on Wikipedia data set.

Approach	Building time (mins)	Memory footprint (GB)	Disk space (GB)
ST2I	**0.8m**	2.9	1.92
RCA	11.9m	19.2	5.9
Lucene	1.25m	**0.7**	**0.56**
SFC-QUAD	222.4m	10.59	1.4

regularly compressing and writing data to disk. ST2I used approximately 2.5 times the size of original data sets, as we use in-place sorting when creating k-d tree. RCA and SFC-QUAD required substantially more memory, especially for large data sets – RCA needs to maintain a spatial structure for every word and SFC-QUAD needs to reassign all object IDs based on their locations on the Z-curve.

Disk Space Usage. In Figure 3c, we show the disk space requirements for each index. All indices show a linear behavior for disk space usage as the data grows. Lucene requires the least amount of disk space. This can be attributed to the fact that Lucene uses LZ4 algorithm to compress data before writing to disk. However, this negatively affects the index construction time as well as the query processing times. SFC-QUAD also uses a compression algorithm called OPT-PFD to store object IDs and word frequencies in blocks, thus, effectively reducing the disk space usage. ST2I used less than half of the space required by RCA.

4.3 Impact of Index Design

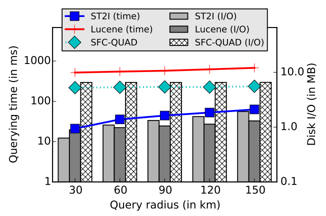
To understand the impact of our design choices, we evaluated each individually and measured the corresponding speedup in index construction, query processing, and space usage.

Text Mapping. In addition to our context-free text mapping algorithm, ST2I can work with any other monotonic function. For instance, a straightforward approach would be to extract all distinct terms, sort them and map each term into its corresponding position in the list. This approach requires an additional full scan of the data, what negatively impacts the index construction time: the time required to build index increases by 51.21% (from 10m17s to 15m37s). Furthermore, this approach is not update friendly: the index to be re-constructed from scratch if data with new terms are added.

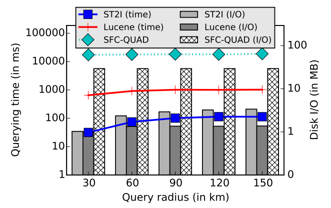
Compactness of Node Data. ST2I uses an implicit splitting strategy and a breadth-first order array, thus, it only needs to store a pointer to the left child in the KNode – the pointer to the right child will always be the subsequent item in the list. When we disable the implicit splitting strategy, each node on KTree needs to store the split value, left and right pointers. As a result, the size of KTree is increased by 50% (from 0.96GB to 1.5GB). The breadth-first order array also helps reduce the processing time for both range queries (36.98%) and top-k queries (23.96%).

Block-Based Data Access. The size and depth of the KTree depend on number of data points B stored in each block. For our experiments, we used $B = 32$. When we disabled the block-based data access, the size of KTree increased 21.87 times (from 0.96GB to 21GB). A larger KTree also leads to slower query processing time (3.78 times slower for range queries and 2.52 times for top-k queries).

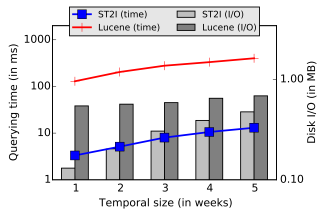
Long Keywords. In order to measure the overhead of additional comparisons for longer keywords, we generated



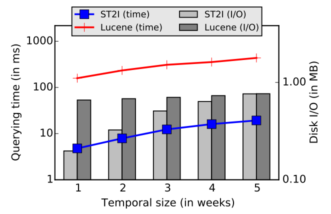
(a) Varying spatial sizes



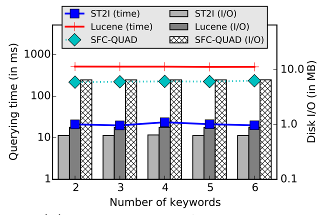
(b) Varying spatial sizes



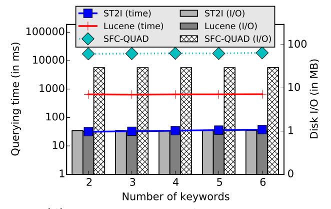
(c) Varying temporal sizes



(d) Varying temporal sizes



(e) Varying # of keywords



(f) Varying # of keywords

Figure 4: Performance of range queries on *EASY* (a, c, e) and *HARD* (b, d, f) workloads on Twitter data sets.

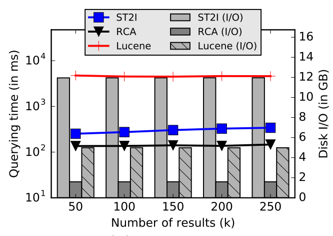
1000 random 2-keyword queries with terms consisting of more than 12 characters and compared the average query processing time of ST2I with its modified version where we used 128-bit integers to encode words. Experiment results showed that query processing time only increased by 1.57%.

4.4 Performance of Range Queries

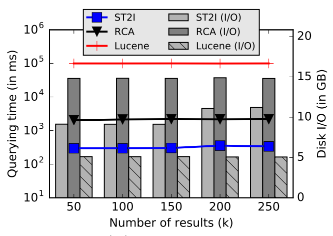
We compared the query evaluation performance of ST2I against Lucene and SFC-QUAD using the Twitter data set. RCA is not included in this evaluation since it was designed for top-k queries.

Queries. We created a workload consisting of 1000 unique queries, where each query contains 2 keywords (combined with AND semantics) and covers a 30km radius. We then created additional workloads by varying the number of keywords and radius size for the queries, as well as different combinations of spatial-temporal popularity levels (Section 4.1). Since SFC-QUAD was not designed to handle temporal constraints, the default workload used to compare all approaches only contains spatial and textual constraints. Different temporal constraints were used to compare ST2I and Lucene.

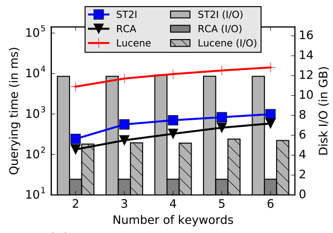
Results. Figure 4 shows results for *EASY* and *HARD* workloads. In general, ST2I performed better than Lucene and SFC-QUAD, especially for *HARD* queries. This is due to the simultaneous filtering over multiple dimensions: queries with both spatial and textual constraints are evaluated considerably faster than spatial or textual alone. Figure 4 also shows results of ST2I, Lucene and SFC-QUAD with different query variations on the whole Twitter data set. Querying times for ST2I are smaller than Lucene and SFC-QUAD by a wide margin. In all experiments, query performance of ST2I is proportional to the number of disk I/Os.



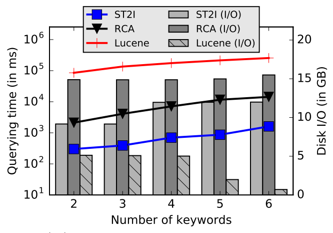
(a) Varying k



(b) Varying k



(c) Varying # of keywords



(d) Varying # of keywords

Figure 5: Performance of top-k queries on *EASY* (a, c) and *HARD* (b, d) workloads on Twitter data sets.

Table 4: Top-k queries on Wikipedia data set.

Approach	*EASY*	*EASY* (I/O)	*HARD*	*HARD* (I/O)
ST2I	**45.570ms**	0.92MB	**50.434ms**	0.88MB
RCA	61.685ms	0.18MB	174.504ms	0.41MB
Lucene	714.365ms	0.36MB	3996.813ms	0.38MB

4.5 Performance of Top-k Queries

We compared query evaluation performance for all approaches that support top-k queries using both the Twitter and Wikipedia data sets. Since RCA does not support temporal constraints, we only generated queries with spatial and textual constraints.

Queries. We created a workload consisting of 1000 unique queries, where each query contains 2 keywords, a location, and default query parameters $k = 50, \gamma = 0.3$. The textual relevance semantic used was **COSINE**. We created additional workloads by varying k and number of keywords.

Results. Figure 5 shows the query processing times for each approach with different query workloads on the Twitter data set. Results on Wikipedia data set are summarized in Table 4. ST2I and RCA show comparable performance on *EASY* workload. On *HARD* workload, ST2I performed much better than other approaches. For instance, ST2I took 170ms on average to return the top 50 tweets, while RCA and Lucene took 2,021ms and 85,324ms, respectively. The results show the benefit of using a single index for spatial, temporal and textual components, and the benefit of simultaneous filtering over multiple dimensions. All other approaches select all objects that match textual constraints before applying spatial constraints. When keywords are popular, the number of objects that matches textual constraints increases exponentially.

Note that in Figure 5, the number of disk I/Os does not correlate with the query processing time. The reason is that the time required for disk I/O access is considerably smaller than the query processing time; disk I/O is not a bottleneck in all approaches (we have validated this information by analyzing the *iowait* times reported by the OS that showed to be negligible during the query executions). Thus, the effect of disk I/O on the overall query processing performance is very small.

5. RELATED WORK

Early works on spatio-temporal textual indexing used separate data structures for space, time and text [30, 32]. However, no performance results were reported. Cozza et al. [15] used PostgreSQL/PostGIS to index geo-tagged tweets. Since PostgreSQL does not provide built-in top-k query processing and the reported performance is several orders of magnitude slower than ST2I, we did not include PostgreSQL in our comparison.

While both spatial and temporal indexes have been extensively studied in the literature, temporal and spatial issues were treated independently. To the best of our knowledge, ST2I is the first attempt to create an integrated spatio-temporal textual index.

Spatial Keyword Index. Numerous spatial keyword indexing strategies have been introduced in the past few years. Chen et al. [12] carried out a comprehensive experimental evaluation of 12 state-of-the-art strategies. In general, these strategies can be categorized into three main types: spatial-first, text-first, and combined. Spatial-first indices organize spatial components into spatial data structures such as IR^2-tree [16], R-tree [11], IR-tree [23], and WIBR-tree [33]. Textual components are inserted into spatial tree nodes or grid cells, in various formats such as inverted lists [11, 23] and inverted bitmaps [16, 33]. Christoforaki et al. [13] used a quadtree for spatial components and space filling curves for document IDs. They also compressed the document ID and object frequencies in inverted lists. A major drawback of spatial-first indices is that they are very sensitive to the spatial constraints. These approaches are not optimized for queries without a spatial constraint or with spatial constraints that have low selectivity, regardless of their textual constraints. ST2I does not suffer from this problem since all constraints types are considered for filtering the results.

Khodaei et al. [22] introduced a combined index called SKIF which stores both spatial and textual components in an inverted list. Spatial components are organized into cells, and each cell is represented by an entry in the inverted list. Unfortunately, this approach is prone to data skew and scalability issues because of the grid structure. In contrast, by adopting a hierarchical space-partitioning data structure, ST2I avoids these issues.

Several text-first indices have been proposed to support top-k spatial keyword queries [12]. Text-first indices use inverted lists to store documents and use spatial data structures for the spatial components. Rocha et al. [29] used an R-tree to store spatial components for frequent keywords and a flat list for infrequent keywords. Zhang et al. proposed the use of a quadtree [35] and Z-order encoding [34]. They also introduced score-bounded access to both textual and spatial components. Similar to spatial-first, text-first indices are not suitable for queries with complex spatial-temporal constraints.

Temporal Keyword Index. A number of approaches have been proposed that support temporal keyword queries. Berberich et al. [8] proposed a technique that supports temporal range queries. Their goal was to provide a better response time than a sequential scan by allowing efficient access to documents within the query time range. Jin et al. [20] proposed several indices based on inverted files, B+-tree and MAP-21 triple index. For each query, they retrieved candidates from each index separately and merged them to produce final results. However none of their ranking functions took into account the temporal relevance score.

Khodaei et al. [21] proposed a temporal-textual index structure called T^2I^2 that combines both temporal and textual components into an inverted list. Textual components are organized into cells and each cell is represented by an entry in the inverted list. As with any grid structure, this approach is also prone to data skew besides having a large storage requirement. He et al. [19] introduced methods to provide better support for querying over versioned documents. They studied different partitioning methods to organize documents by time. It is not clear how these specialized indices can efficiently support spatial constraints.

6. CONCLUSIONS

In this paper, we introduced ST2I, a new indexing strategy that uniformly handles text, space and time in a single structure. ST2I uses context-free text mapping algorithms to encode words in text as numbers, and a block-based kd-tree structure embedded in a breadth-first order layout that has a space requirement up to 50% smaller than current techniques. This effectively increases the memory locality, leading to better querying performance and enabling larger data sets to be indexed. The flat memory structure also allows us to take advantage of 64-bit memory mapped files to significantly reduce overhead for serving queries from disks. Our experimental results show that ST2I performs uniformly better than the other approaches: it is substantially faster for both index construction and query evaluation, and it also has smaller memory footprints and storage requirements than most approaches.

In future work, we intend to improve performance of ST2I by exploiting the parallelism supported by multiple CPU cores, exploring efficient update approaches and developing a variable-bitrate encoding to support longer terms and larger character set.

Acknowledgments. This work was supported in part by NSF award CNS-1229185 and DARPA award FA8750-14-2-023. Opinions, findings and conclusions expressed in this material are those of the authors and do not necessarily reflect the views of the NSF or DARPA.

7. REFERENCES

[1] Lucene. http://lucene.apache.org, 2014.

[2] nanoflann: Approximate Nearest Neighbor. https://code.google.com/p/nanoflann/, 2014.

[3] Spatial C++ Library. http://spatial.sourceforge.net, 2014.

[4] WolframAlpha. http://www.wolframalpha.com/input/?i=average+english+word+length, 2014.

[5] L. Barbosa, K. Pham, C. Silva, M. R. Vieira, and J. Freire. Structured Open Urban Data: Understanding the Landscape. *Big Data*, pages 144–154, 2014.

[6] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. In *SIGMOD*, pages 322–331, 1990.

[7] J. L. Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *CACM*, pages 509–517, 1975.

[8] K. Berberich, S. Bedathur, T. Neumann, and G. Weikum. A Time Machine for Text Search. In *SIGIR*, 2007.

[9] S. Berchtold, D. A. Keim, and H.-P. Kriegel. The X-tree: An Index Structure for High-Dimensional Data. In *VLDB*, pages 28–39, 1996.

[10] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan. Time Bounds for Selection. *J. Comput. Syst. Sci.*, pages 448–461, 1973.

[11] A. Cary, O. Wolfson, and N. Rishe. Efficient and Scalable Method for Processing Top-k Spatial Boolean Queries. In *SSDBM*, volume 6187, pages 87–95. 2010.

[12] L. Chen, G. Cong, C. S. Jensen, and D. Wu. Spatial Keyword Query Processing: An Experimental Evaluation. *PVLDB*, pages 217–228, 2013.

[13] M. Christoforaki, J. He, C. Dimopoulos, A. Markowetz, and T. Suel. Text vs. Space: Efficient Geo-search Query Processing. In *CIKM*, pages 423–432, 2011.

[14] R. Chunara, J. Andrews, and J. Brownstein. Social and news media enable estimation of epidemiological patterns early in the 2010 haitian cholera outbreak. *AJTMH*, 2012.

[15] V. Cozza, A. Messina, D. Montesi, L. Arietta, and M. Magnani. Spatio-temporal keyword queries in social networks. In *ADBIS*, volume 8133, pages 70–83. 2013.

[16] I. De Felipe, V. Hristidis, and N. Rishe. Keyword Search on Spatial Databases. In *ICDE*, pages 656–665, 2008.

[17] R. Finkel and J. Bentley. Quad trees a data structure for retrieval on composite keys. *Acta Informatica*, 1974.

[18] A. Guttman. R-trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD*, pages 47–57, 1984.

[19] J. He and T. Suel. Faster temporal range queries over versioned text. In *SIGIR*, pages 565–574, 2011.

[20] P. Jin, H. Chen, X. Zhao, X. Li, and L. Yue. Indexing temporal information for web pages. *ComSIS*, 2011.

[21] A. Khodaei, C. Shahabi, and A. Khodaei. Temporal-Textual Retrieval: Time and Keyword Search in Web Documents. *IJNGC*, 2012.

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

[23] Z. Li, K. Lee, B. Zheng, W.-C. Lee, D. L. Lee, and X. Wang. IR-Tree: An Efficient Index for Geographic Document Search. *TKDE*, pages 585–599, 2011.

[24] D. B. Lomet and B. Salzberg. The hB-tree: A Multiattribute Indexing Method with Good Guaranteed Performance. *TODS*, pages 625–658, 1990.

[25] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to information retrieval*, volume 1. Cambridge University Press, 2008.

[26] O. Procopiuc, P. Agarwal, L. Arge, and J. Vitter. Bkd-Tree: A Dynamic Scalable kd-Tree. In *SSTD*, volume 2750, pages 46–65. 2003.

[27] P. Rigaux, M. Scholl, and A. Voisard. *Spatial Databases with Application to GIS*. Morgan Kaufmann Publishers Inc., 2002.

[28] J. T. Robinson. The K-D-B-tree: A Search Structure for Large Multidimensional Dynamic Indexes. In *SIGMOD*, pages 10–18, 1981.

[29] J. a. B. Rocha-Junior, O. Gkorgkas, S. Jonassen, and K. Nørvåg. Efficient Processing of Top-k Spatial Keyword Queries. In *SSTD*, pages 205–222, 2011.

[30] J. Strötgen and M. Gertz. TimeTrails: A System for Exploring Spatio-temporal Information in Documents. *PVLDB*, pages 1569–1572, 2010.

[31] Twitter. https://about.twitter.com/company, 2014.

[32] B. Wang, H. Dong, A. P. Boedihardjo, C.-T. Lu, H. Yu, I.-R. Chen, and J. Dai. An Integrated Framework for Spatio-temporal-textual Search and Mining. In *SIGSPATIAL*, pages 570–573, 2012.

[33] D. Wu, M. L. Yiu, G. Cong, and C. S. Jensen. Joint Top-K Spatial Keyword Query Processing. *TKDE*, 2012.

[34] D. Zhang, C.-Y. Chan, and K.-L. Tan. Processing Spatial Keyword Query As a Top-k Aggregation Query. In *SIGIR*, pages 355–364, 2014.

[35] D. Zhang, K.-L. Tan, and A. K. H. Tung. Scalable Top-k Spatial Keyword Search. In *EDBT*, pages 359–370, 2013.

[36] P. Zhou and B. Salzberg. The hB-pi* Tree: An Optimized Comprehensive Access Method for Frequent-Update Multi-dimensional Point Data. In *SSDBM*, 2008.