STILT: Unifying Spatial, Temporal and Textual Search using a Generalized Multi-dimensional Index

Yoann Arseneau, Saransh Gautam, Bradford G. Nickerson and Suprio Ray
University of New Brunswick, Fredericton, Canada
{yoann.arseneau,sgautam,bgn,sray}@unb.ca

ABSTRACT

The proliferation of location-enabled sensors, smart phones, and the power of digital messaging combined with social media platforms is producing a deluge of multi-dimensional data. Novel index structures are needed to efficiently process massive amounts of geotagged data, and to promptly answer queries with textual, spatial, and temporal components. Existing approaches to spatio-temporal text indexing lack a unified index supporting efficient range and top-k search on any combination of location, time, or text.

We introduce a generalized, multi-dimensional index called Spatio-temporal Textual InterLeaved Trie (STILT), which unifies spatial, textual, and temporal components within a single structure. STILT is a general-purpose index supporting subset searches (e.g. spatio-textual, spatio-temporal, spatial, textual), as well as full spatio-temporal textual searches. STILT uses a binary trie-based index interleaving text, location, and time information in a space-efficient manner. STILT supports parallel building of the index and concurrent execution of spatio-temporal textual queries, including top-k and range search queries. With extensive evaluation, we demonstrate that STILT is significantly faster than state-of-the-art approaches in terms of index construction time and search latency.

CCS CONCEPTS

• **Information systems → Spatial-temporal systems**; **Data access methods**; **Information retrieval query processing**.

KEYWORDS

spatial, spatio-textual, spatio-temporal, multi-dimensional index

ACM Reference Format:

Yoann Arseneau, Saransh Gautam, Bradford G. Nickerson and Suprio Ray. 2020. STILT: Unifying Spatial, Temporal and Textual Search using a Generalized Multi-dimensional Index. In *32nd International Conference on Scientific and Statistical Database Management (SSDBM 2020), July 7–9, 2020, Vienna, Austria.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3400903.3400927

1 Introduction

The big data era is characterized by a rising volume of data generated from a variety of sources. These data sources include mobile devices and sensors, web and social media, internet-of-things communication, enterprise applications, digital archives and public records. A growing number of these mobile devices are geolocation-enabled. The information produced by these sources often contain spatial, temporal, and textual components. A notable example of this is social media data, such as Twitter, Facebook, and Instagram posts. Blog posts and articles posted online are also rich sources of data containing time, location, and text. Other datasets may have explicit location information (e.g. Wikipedia articles), or some spatial components that can be used to derive a location. Invariably, all data has a time component, whether it be creation time, access time, or some other data-specific time.

These datasets, involving text, time, and spatial information can provide valuable insights and offer governments and businesses an edge. Indexing is a common technique to support efficient query processing, particularly when the data volume is large and growing fast. A significant body of research exists on the topic of indexing temporal data. To index spatial data, tree-based approaches such as the R-tree, quadtree, and k-d tree are the most widely adopted. Supporting text search became an important area of research as search engines like google and yahoo became popular. The inverted index and its variants are the most popular data structure used for full text search in information retrieval systems and search engines. As Location-Based Services (LBS) became widespread in the last decade, the research community focused on developing combined spatio-temporal indexes. Consequently, a number of systems were proposed, such as the B^x-tree [9], ST2B-tree [3], STR-tree [21] and MV3R-tree [25]. The rise of social media provided an impetus to invent indexes that can combine spatial and textual search. Indexing approaches such as the IR-tree [13] and the DIR-tree [6] are the outcome of this line of research. These indexes normally integrate an inverted index with a tree based approach, primarily an R-tree or quadtree, and take either a spatial-first or text-first approach in applying the search constraints. Additional optimizations provided by S2I [23] and I^3 [30] take advantage of the fact that some keywords appear far more frequently than others.

As more and more data sets include text, time, and location components in the same document object, it is becoming increasingly important to support queries that specify search criteria based on all 3 components. An example query is *"Retrieve the 10 most relevant documents that contain the keywords 'midterm' and 'election' that were posted between November 6, 2018 and November 30, 2018 within 50 km of Washington, DC"*. To answer such a query, a spatio-temporal index can be used to apply the location and time criteria and an inverted index can be used to search based on the text. Then an intersection of these two resulting sets would yield the result set satisfying all 3 constraints. Another way to process

SSDBM 2020, July 7–9, 2020, Vienna, Austria

ACM ISBN 978-1-4503-8814-6/20/07…$15.00
https://doi.org/10.1145/3400903.3400927

Table 1: Some STILT index variants and their corresponding path schedules.

STILT variants	Description	b	L	Path schedule
$\text{STILT}_{\text{stx}}$	Spatio-temporal textual	16	64	$y_1x_1w_1t_1y_2x_2w_2t_2 \ldots y_bx_bw_bt_b$
STILT_{sx}	Spatio-textual	21	63	$y_1x_1w_1y_2x_2w_2 \ldots y_bx_bw_b$
STILT_{st}	Spatio-temporal	21	63	$y_1x_1t_1y_2x_2t_2 \ldots y_bx_bt_b$

these queries would require using a spatio-textual index first, followed by a temporal index to filter out results that do not match the query time range. These methods, however, necessitate employing more than one index and hence can be costly to process spatio-temporal keyword queries. The research community, however, did not pay much attention to this problem. ST2I [8] is the first index we are aware of that provides a unified index and integrated ranking scheme that handles space, time, and text in a single data structure. ST2I, however, has some limitations, including support for a limited character-set, lack of support for parallel operations to take advantage of modern multi-core processors, and the need for rebuilding the index when new data needs to be inserted. Note that a few other systems exist that support queries on data involving the spatial, temporal, and keyword attributes; however, they do not use an integrated data structure or a unified spatio-temporal textual ranking scheme like ST2I. For instance, Taghreed [15] supports a few types of queries such as spatio-temporal boolean range keyword search, and top-k frequent keyword query, but it does not support a ranking scheme that considers both spatial and textual relevance. Mercury [16] is designed to support top-k spatio-temporal textual queries. Its ranking mechanism, however, only incorporates spatial and temporal relevance, not textual relevance.

While it is important to efficiently support spatio-temporal textual queries using a single unified index, it could be argued that such queries are not common and hence they do not warrant a specialized index. The same argument can be made for queries involving a subset of multiple dimensions (eg. temporal-textual or spatio-textual). Therefore, it is necessary to develop a **generalized index** that can support queries with any combination of dimensions involving location, time and text. To address the issues with existing approaches, we introduce a generalized, integrated, multi-dimensional index that we call Spatio-temporal Textual Interleaved Trie (STILT). It is based on a multi-dimensional binary Patricia trie [18]. Tries are highly efficient on modern hardware for in-memory indexing [12]. Other researchers [2] have found that well-designed tries can outperform comparison based indexes (e.g. B-tree or R-tree, and their variants), particularly with string data. STILT converts indexable components (e.g. spatial, textual, temporal) to integers via a mapping function for each dimension. These integers are then bit-interleaved according to a schedule that dictates the order in which the bits are concatenated to produce an interleaved bit-string that we call a *path schedule*. Depending on the dimension(s) being indexed, a different path schedule is utilized by the STILT index, as shown in Table 1. Each letter in the suffix in each variant indicates the supported dimension, where letters *s*, *t* and *x* stand for **s**patial, **t**emporal and te**x**tual dimensions, respectively. For instance, STILT_{sx} supports indexing spatial and textual dimensions. Here, y_1, x_1, w_1, t_1 refer to the first bit of the latitude, longitude, word, and time, respectively, and b is the number of bits for each of the dimensions. L represents the length of the path schedule which is the maximum height of the Patricia trie. $\text{STILT}_{\text{stx}}$ is effectively a four-dimensional Z-order curve. In Section 4.2 we discuss our index and path schedule in depth. Figure 4 shows an example $\text{STILT}_{\text{stx}}$ index for the documents in Table 2 (with $b = 8$ and $L = 32$).

To our knowledge, STILT is the first **generalized, single-structure** index that supports fully integrated spatio-temporal textual searches, as well as subset dimensional searches involving any combination of dimensions (e.g. spatio-temporal, spatio-textual, spatial, textual). Note that unlike STILT, ST2I only supports integrated spatio-temporal textual searches, but not subset dimensional searches. We conducted a comprehensive experimental evaluation of STILT with 3 real-world datasets (Spaten, Tw20mi and Wikipedia: see Table 4 for details). To demonstrate that STILT is capable of efficiently supporting full spatio-temporal textual and subset dimensional searches, we evaluated STILT against several different indexes. To compare spatio-textual query performance, we evaluated STILT (STILT_{sx}) against a fast spatio-textual index I^3. Our results demonstrate that STILT_{sx} is one order of magnitude faster than I^3 with spatio-textual top-k search. To compare spatio-temporal query performance, we conducted experiments with a B^{x}-tree index, which was reported to be one of the fastest spatio-temporal index in [4]. STILT_{st} significantly outperforms B^{x}-tree on spatio-temporal range queries. To compare STILT ($\text{STILT}_{\text{stx}}$) on integrated spatio-temporal textual features, we chose ST2I, which is the state-of-the-art integrated spatio-temporal textual index. Experimental results suggest that $\text{STILT}_{\text{stx}}$ is significantly faster than ST2I in terms of index construction and query (range and top-k search) execution times. For instance, with the Wikipedia dataset, the index construction time of STILT is 3.3× faster than ST2I. Furthermore, range search execution time of STILT is up to 18.2× faster, and top-k search execution time is up to 11.6× faster than ST2I. The key contributions of this paper are as follows:

- We developed a novel, generalized in-memory spatio-temporal textual index STILT, based on a multi-dimensional binary trie, which incorporates a number of optimizations to attain space and query efficiency.
- Our system supports parallel index building and concurrent query execution, exploiting modern multi-core machines.
- In our experimental evaluation involving 3 real-world datasets, STILT (variants) performed significantly better than ST2I, I^3 and B^{x}-tree indexes.

The rest of the paper is organized as follows. In Section 2 we present related work. Problem definitions for range and top-k search are presented in Section 3. We introduce our index and present an illustration in Section 4. The index algorithms, including index construction and search, are discussed in Section 5. In Section 6 we present analyses of storage and algorithm costs. An experimental evaluation is presented in Section 7 and finally, we draw conclusions in Section 8.

2 Related Work

First, we present related work on trie-based indexes. Then, we discuss prior research pertaining to spatio-temporal and spatio-textual indexes followed by previous work on integrated spatio-temporal textual indexes.

Note that STILT supports ranked *ad hoc (snapshot)* query processing in a modern server class machine. Therefore, distributed spatio-textual indexing approaches, such as Tornado [17], or publish-subscribe indexing approaches for continuous queries like AP-Tree [27] are beyond the scope of our paper.

2.1 Trie-based Index

Tries are data structures in which all the children of a node share a common prefix and the search process involves determining which child to proceed with depending on the remaining components of a search key. In recent times trie-based main memory indexes have received renewed attention, particularly with the popularity of in-memory databases. The Adaptive Radix Trie (ART) [12] is a recently proposed index that dynamically adapts the node structure by selecting more compact representations. A follow-up work, Height Optimized Trie (HOT) [2], attempts to optimize the tree height by combining multiple nodes of a binary trie into compound nodes.

Like ART and HOT, our STILT index is also a trie-based approach. STILT, however, is based on a binary Patricia trie [18], where interior nodes contain a sequence of bits common to all the interior node's children called the *skip bit list*. For tries indexing random keys, it has been shown [11] that search of Patricia tries visits fewer nodes than two other types of multi-dimensional tries. STILT is a multi-dimensional binary Patricia trie that adaptively chooses the most compact node representation. This approach provides a highly compact representation generalizable to any number of required dimensions.

2.2 Spatio-temporal Index

Due to the rising popularity of Location-Based Services (LBS), a number of spatio-temporal indexes were proposed. The BB^x-index [14] is a technique that keeps a forest of B^x-trees [9], each tree for a different time interval. A B^x-tree [9] exploits a B-tree to index the object locations converted into space filling curve (SFC) codes. The ST2B-tree [3] is another B-tree based technique.

Several R-tree based approaches have also been proposed. The STR-tree [21] uses an R-tree to index moving object trajectories and hence can perform poorly for long trajectories. The MV3R-tree [25] uses multiple versions of R-trees.

2.3 Spatio-textual Index

Spatio-textual indexes can be classified into three groups based on their structure: R-tree based, grid-based and space filling curve based. The R-tree based approaches generally combine an R-tree with an inverted file. The IR-tree [6] is a classic example of this. To deal with frequent terms and infrequent terms, the S2I [23] index uses different approaches. The I^3 index [30] utilizes a textual partitioning approach similar to S2I. However, their spatial data structure differs from that of S2I, as they use a quadtree instead of an R-tree. SFC-QUAD [5] is a space filling curve approach that orders documents on their position in a Z-order curve and stores them in an inverted list. The grid-based indexes normally integrate a grid index with a textual index. These include ST & TS [26] and SKIF [10]. Researchers have attempted to incorporate time as part of spatio-textual query processing. These are not unified spatio-temporal textual indexes, in the sense that the temporal dimension is not integrated in the indexing structure as in a spatio-temporal index, but rather it is used as a filter for temporal pruning. The authors in [22] proposed a hybrid index that integrates a grid with an STR R-tree. It supports spatio-textually ranked top-k search that returns the most recent documents. Recently, Almaslukh et al. [1] evaluated ten main-memory index structures. Their temporal boolean range query (TBRQ) used a ranking scheme based on spatial and temporal components, but without a textual component.

2.4 Unified Spatio-temporal Textual Index

According to the authors [8], ST2I is the first index that integrates spatial, temporal and textual components in a single structure. However, ST2I has several limitations that include lack of support for non-Latin character sets, the need for complete index reconstruction in order to insert new data, and a lack of support for concurrent index building and query execution. ST2I uses a k-d tree as the basis for its index, which requires median finding during insertion to guarantee a balanced tree supporting efficient search. This limits the use of ST2I to relatively static environments where new data arrives slowly, such as historical archives. In contrast, STILT supports concurrent insertion and search, and is ideal for highly dynamic environments where new documents are arriving continuously.

Other than ST2I, a few indexes were designed to support queries with spatial, temporal, and textual attributes. They include Taghreed [15] and Mercury [16]. Unlike ST2I, they neither integrate space, time and text in a unified structure, nor support combined spatio-temporal textual ranking.

3 Definitions

In this section we define range and top-k search for combined spatio-temporal textual queries. Queries involving a subset of these dimensions can be defined similarly, but are omitted due to space constraints.

Definition 3.1. Given a set of documents O, spatio-temporal textual *range search* finds and reports the set C of document identifiers matching a query q as follows:

$$\begin{aligned} C \equiv \text{docindex}(\{&\forall c \in C : |q.W \cap c.W| > 0, \\ &q.\mathcal{L}.y_L \le o.\ell.y \le q.\mathcal{L}.y_U, q.\mathcal{L}.x_L \le o.\ell.x \le q.\mathcal{L}.x_U, \\ &q.\mathcal{T}_L \le o.t \le q.\mathcal{T}_U\}) \end{aligned} \quad (1)$$

Each document $o = (id, \ell, W, t)$ in the set O of indexed documents to be searched has an id, a location $\ell = \{\ell.y, \ell.x\}$, a set of words W describing the document, and a document time t (e.g. the time and date the document was inserted into the database). Further, $q = (\mathcal{L}, W, \mathcal{T})$, where $\mathcal{L} = \{[y_L, y_U], [x_L, x_U]\}$ is the set of two dimensional spatial range intervals defining a search rectangle, W is the set of words being searched for, and $\mathcal{T} = [\mathcal{T}_L, \mathcal{T}_U]$ is the time range of the query. docindex() returns the document identifiers of the search results.

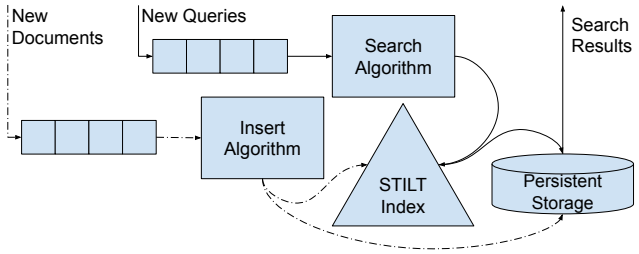

Figure 1: System architecture.

The above definition corresponds to a Boolean "OR" search for queries with $|q.W| > 1$; i.e. only one of the keywords in $q.W$ needs to be present in documents matching the query. A Boolean "AND" query (where documents in the range search results must include all keywords in $q.W$) is also supported by STILT. STILT supports *partial* range search, meaning that any of the query bounds (including query keywords) can be ignored. For example, a range search to find all documents matching a query with $o.t \geq$ "2019-10-30 00:00" can specify the time range as $\mathcal{T} =$ ["2019-10-30 00:00", $*$], where the $*$ symbol means to ignore the upper bound, and search for all documents with $o.t \geq q.\mathcal{T}_L$.

Definition 3.2. A spatio-temporal textual *top-k search* reports the set D of the k highest ranked documents matching a query q. A ranking scheme is used to assign scores and order the documents in D. The query $q = (\ell, W, t, k)$ defines a query point $q.\ell$, a query keyword set $q.W$, a query time $q.t$, and a parameter k where k is the number of ranked documents to return.

$$D \equiv \text{sort}(\{S(q,d) : \forall d \in E, |D| \leq k\}) \tag{2}$$

Here, $S(q,d)$ is the combined spatio-temporal textual score calculated with Equation 7, which is based on a unified ranking scheme, explained in Section 5.4. This ranking scheme combines spatial, temporal and textual components into a single score $\in [0,1]$. D is a sorted set of size k or less (if fewer than k documents match the query) consisting of the (up to) k highest scoring documents matching Equation 2. We use a priority queue to maintain the sorted set D of the (up to) k highest ranked documents while searching, as described in Algorithm 7. E is the result of a partial search with query $q = (*, W, *, k)$ which searches all documents in O with keywords matching W to find the *top-k* matches based on the score $S(q,d)$.

4 System Organization

We present our Spatio-temporal Textual Interleaved Trie (STILT) index and describe the system organization in this section.

STILT is a generalized multi-dimensional index designed to support efficient insertion and search. STILT utilizes an adaptive node allocation strategy to minimize memory usage (see Section 5 for further details). STILT exploits multi-threading to support efficient insertion and search with modern multi-core architecture machines.

4.1 System Organization

Our overall system is composed of an in-memory index, STILT and a persistent storage based on the LSM-tree [20] to store documents, as shown in Figure 1. Every document is assigned a unique *id* upon insertion into the system. The LSM-tree maps *id*s to documents, while the index maps keys to *id*s. A document is defined as $o = (id, \ell, W, t)$, where id is the document identifier, $\ell = (y, x)$ is its spatial location, W is the ordered list of words for the document, and t is the document's timestamp. A document has $|W|$ keys, defined as $(o.id, o.\ell, w, o.t, \texttt{extra}(o,w)), \forall w \in \texttt{unique}(o.W)$. The `unique` function returns a set of unique words in the sequence, after converting them to lower case. The `extra` function returns any metadata required by the index. To support top-k queries, `extra` returns $|w \cap o.W| \div |o.W|$ for efficient textual-relevance scoring.

4.2 Index Structure

The STILT index is essentially an in-memory data structure based on binary Patricia trie [18], which maps N-dimensional keys to document ids. It requires a predetermined maximum length $L \in \mathbb{Z}^+$, a number of dimensions $N \in \mathbb{Z}^+$, a mapping function for each dimension, and a *path schedule*. We discretize the space into 2^L cells, each of which corresponds to a leaf node of the trie. To map a key to a cell, we map each of its components to an integer, then we combine these integers into a single string of L bits we call a *path*. The combining process interleaves the bits according to a *path schedule* (see Table 1), which dictates the order in which the bits are concatenated to produce a path. The schedule also implicitly dictates how many significant bits are needed for each integer. A path maps to a cell by interpreting its bits as left or right choices (i.e. 0 = go left, 1 = go right) as we descend from the root node. Multiple keys share the same leaf if their paths are identical. The notion of a path schedule enables STILT to be a generalized multi-dimensional index, as different index variants (as shown in Table 1) can utilize different path schedules. Our index can thus support subset dimensional searches involving any combination of dimensions. For instance, index variant $\text{STILT}_{\text{stx}}$ enables unified spatio-temporal textual searches, whereas STILT_{sx} supports spatio-textual search and STILT_{st} supports spatio-temporal search. For the sake of simplicity, we primarily discuss the most general variant $\text{STILT}_{\text{stx}}$ in this and the next section.

In all of our implementations of STILT, we use the same mapping functions. For spatial and temporal dimensions, we use a simple linear mapping based on a bounding range, which includes all data points. For example, the Twitter and Spaten datasets' spatial bounding rectangle includes the Contiguous United States, while the Wikipedia dataset's includes the entire globe. For keywords, we use a case-insensitive hash function which converts the characters of the keyword into 5-bit strings as described in Algorithm 1. Notice that latin-alphabet characters map to $[1\ ..\ 27]$, null characters (padding for short words) map to 0, and undefined characters map to 31. The remaining range of $[28\ ..\ 30]$ was arbitrarily assigned to common characters in the datasets.

As illustrated in Figure 2, the STILT index is implemented using two base types: `Node` and `Key`. The `Node` type represents non-leaf nodes in the tree. We do not have a direct representation for leaf nodes; we simply refer directly to the leaf data, which is either a `Key` object or an array of `Key` objects. The `Key` type generalizes the concept of keys as described in Section 4.1. Keys can be of type `FlatKey` or `SlimKey`, depending on which is more space efficient.

4.3 Data Ingestion

The insertion of a document o has three steps. First, we atomically generate a unique positive integer $id \in \mathbb{Z}^+$. Second, we insert the

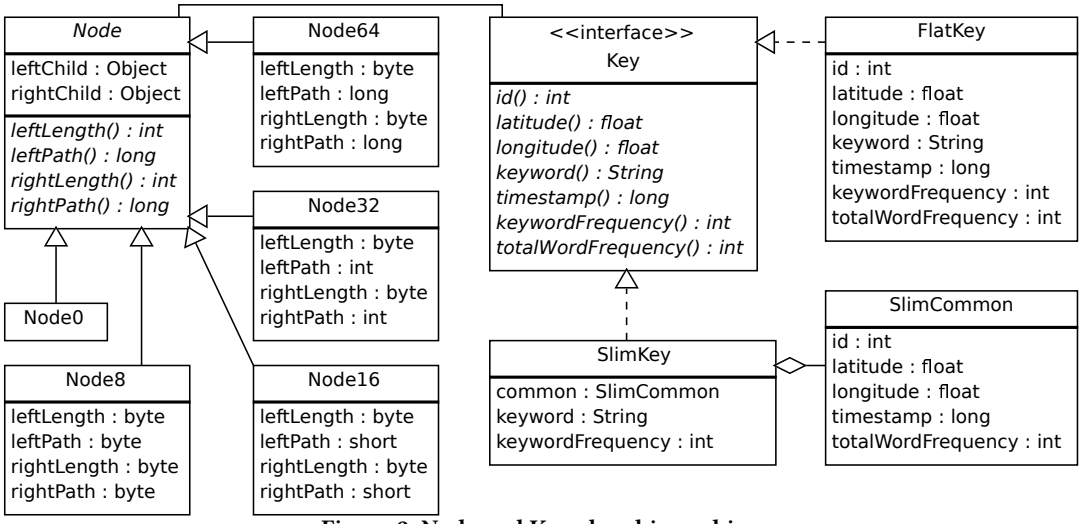

Figure 2: Node and Key class hierarchies.

Algorithm 1: Convert a character to a bit string.

Input : A string s and a number of bits b.

```
1  Function keywordToBits(s, b) : int
2      if length(s) * 5 < b then make string long enough
3          s ← pad_with_null(s, b/5 + 1)
4      bits ← 0
5      for i ← 0 to b/5 do append full characters
6          bits ← (bits << 5) + charToBits(s[i])
7      rem ← b mod 5
8      if rem ≠ 0 then append partial character
9          bits ← bits << rem
10         bits ← bits + (charToBits(s[b/5+1]) >> (5 − rem))
11     return bits << (bit_length(int) − b)
12 Function charToBits(c) : int
13     if c ≥ 'a' and c ≤ 'z' then return c − 'a' + 1
14     else if c ≥ 'A' and c ≤ 'Z' then return c − 'A' + 1
15     else
16         switch c do
17             case '\0' do return 0
18             case '?' do return 27
19             case '“' do return 28
20             case '"' do return 29
21             case '°' do return 30
22             otherwise do return 31
```

mapping ($id \rightarrow o$) into the persistent storage. Third, for each of the $|W|$ 4-dimensional keys $\in o$, we insert the mapping (key $\rightarrow id$) into the index (line 5). The second and third operations must happen in-order to prevent the index from ever referring to a non-existent document.

4.4 Query Processing

STILT supports efficient spatio-temporal textual range search (*Definition 3.1*) and top-k search (*Definition 3.2*). Range search (explained in section 5.3) is implemented by performing a range search on the STILT index to get the *id* of all matching keys, then fetching all unique *id*s from the persistent storage. Top-k search is implemented by fetching all matching keys, grouping them by document *id*, and calculating a score for each *id*, as explained in section 5.4. As a result, we only need to fetch at most k documents from the persistent storage, drastically reducing the number of I/O operations.

Algorithm 2: Insert a document into the system.

Input : A persistent storage LSM, an index index, and a document doc.

Result : Generates a unique id for the document, adds the id → doc mapping to the backing store, and indexes the id.

```
1 Function insert_system(LSM, index, doc) : void
2     id ← new_id()
3     insert(LSM, id → doc)
4     foreach key in doc do
5         insert(index, key → id)
```

Note that our implementation supports concurrent insertion and concurrent searching. It also supports searching while inserting, but in this case the search is not fully isolated. A search is guaranteed to find anything inserted before the search is initiated, but may also include any document whose indexing began before the search terminated. For streaming data applications this is largely inconsequential. If necessary, synchronizing search and insert operations can be achieved by adding a master read/write lock to the index.

4.5 Illustration

Table 2 shows three example documents containing six keys (arising from the two words in each document) along with the binary representation of the first eight bits of each dimension of the key. Colours red, green, blue and purple are used to indicate the latitude, longitude, text and time elements, respectively, of the 4-dimensional key. For this illustrative example, the maximum depth of the Patricia trie is restricted to 32 bits. This provides eight bits for each of the

Algorithm 3: insert a key into the STILT index.

```
1  Function stilt_insert(node, key, id)
2      path ← path_of(key)
3      repeat
4          acquire_lock(node)
5          edge ← pick_edge(node, path)
6          if edge does not exist then create it
7              node_next ← new LeafNode
8              put_edge(node, path, node_next)
9              node ← node_next
10             break
11         if edge matches path then continue search
12             node ← edge.child
13         else split the edge
14             node ← split(edge, path)
15             edge ← pick_edge(node, path)
16         path ← path − edge.path
17         release_lock(node)
18     until path is empty
19     insert(node, (key → id))
```

four dimensions of a key. Our actual implementation of $\text{STILT}_{\text{stx}}$, as mentioned in Section 4.2, has a maximum path length L of 64, providing 16 bits for each of the four dimensions.

Figure 3 illustrates building a spatio-temporal textual $\text{STILT}_{\text{stx}}$ index by inserting the keys in Table 2. The key elements are converted to positive integers via a linear mapping, as explained in section 4.2, with $b = 8$, $L = 32$, t_{min} = `00:00:00`, t_{max} = `23:59:59`, and spatial coordinates $\in [0, 100]$.

Figure 3 also shows the node types `Node32` and `Node8` created as the keys in Table 2 are inserted in the order shown in Table 2. As indicated in sections 4.2 and 5, non-leaf nodes are created with paths of varying length depending on the number of bits required to hold the path to a leaf at the time of creation. This adaptive approach saves space as the STILT index grows, and a larger number of deeper nodes are created. Figure 4 shows the final $\text{STILT}_{\text{stx}}$ index with all edge labels for this example after a compression step that replaces nodes created as type `Node32` with `Node16` and `Node8` node types.

5 Index Algorithms

In this section we discuss the index construction algorithm first. Then we present query processing algorithms.

5.1 Index Construction

STILT is built by iterative insertion of (key → *id*) mappings. The process has three steps, as described in Algorithm 3. We first calculate the path for the key (line 2) by combining the hashes according to the schedule. Then, we iteratively descend (line 3) the trie until we either find (line 12) or create (line 7) the desired leaf node. Finally, we append the (key → *id*) mapping to the leaf node (line 19).

Algorithm 4 describes the process of splitting an edge. It involves two steps. First, we create a common node at the depth at which the existing node and the new path diverge (line 3). Then, we create an edge on the common node to which we attach the preexisting

Algorithm 4: Split an edge by inserting a new node along its length. Mutates its first argument.

```
1  Function split (edge, path) : LeafNode
2      (node_o, length_o, path_o) ← edge
       // create a common node
3      length_c ← clz(xor(path, edge.path))
4      node_c ← new Node
5      edge.node ← node_c
6      edge.length ← length_c
       // re-link original node
7      edge_o ← new Edge
8      edge_o.node ← node_o
9      edge_o.length ← length_o − length_c − 1
10     edge_o.path ← path_o << (length_c + 1)
11     set_edge(node_c, not msb(path), node_o)
12     return node_c
```

node such that its path remains unchanged (line 7). We return the common node (line 12) so *stilt_insert* (Algorithm 3) can populate its empty edge.

5.2 Deduplication

Achieving storage efficiency is an important consideration for our index. To reduce storage requirements, we developed a deduplication strategy for keys. As can be seen in Table 2, each document has a single id, spatial coordinates, timestamp, and a single list of words W. As such, the *id*, *latitude*, *longitude*, *timestamp*, and *total word frequency* fields are identical in each key of a document that has multiple unique words. Instead of storing all the fields in every key using the `FlatKey` node structure (see Figure 2), we create one common object (`SlimCommon`) to store the repeating fields. We then create one unique object (`SlimKey`) to store the *keyword* and *keyword frequency* fields, and a reference to the `SlimCommon` object. In our Java-based implementation, given a number of documents $n = |O|$, each with an average number m of unique words, the total bytes of memory usage using `FlatKey` objects only versus using `SlimCommon` objects only can be expressed as $48mn$ and $40mn+24m$, respectively as shown in Table 3. The memory saving can therefore be expressed as $m(8n - 24)$ bytes. For the datasets we used for evaluation (see Table 4), the storage savings from deduplication amounted to 447.5, 909.7, and 1,656.2 MiB for the Spaten, Tw20mi, and Wikipedia datasets, respectively.

In addition to the deduplication of document keys in the index, we use other storage saving strategies. For instance, we utilize string interning [24] to reduce memory usage, which stores only one copy of each distinct string value.

5.3 Range Search

Our spatio-temporal textual range search algorithm is presented in Algorithm 5. The query is defined by a 2-dimensional spatial range $\mathcal{L}$, a set of words W, and a time range $\mathcal{T}$ (*Definition 3.1*). The search query $(\mathcal{L}, W, \mathcal{T})$ is applied to each keyword in the query i.e. $\forall w \in W$ (line 3). Searching the system for documents matching $(\mathcal{L}, W, \mathcal{T})$ consists of three steps. First, we find every leaf node N which intersects the rectangle defined by $\mathcal{L}$ and may contain at least

Table 2: Three example documents and their binary representation.

Id	Document	Keys	Binary representation
0	(42,96) {desserts,bakery} 18:30:48	(42, 96, desserts, 18:30:48) (42, 96, bakery, 18:30:48)	(01101011,11110101,00100001,11100101) (01101011,11110101,00010111,11100101)
1	(39,72) {alcohol,bar} 18:35:24	(39,72,alcohol,18:35:24) (39,72,bar,18:35:24)	(01100011,10111000,00001011,11100110) (01100011,10111000,00010000,11100110)
2	(41,69) {desserts,diner} 18:35:42	(41,69,desserts,18:35:42) (41,69,diner,18:35:42)	(01101000,10110000,00100001,11100110) (01101000,10110000,00100010,11100110)

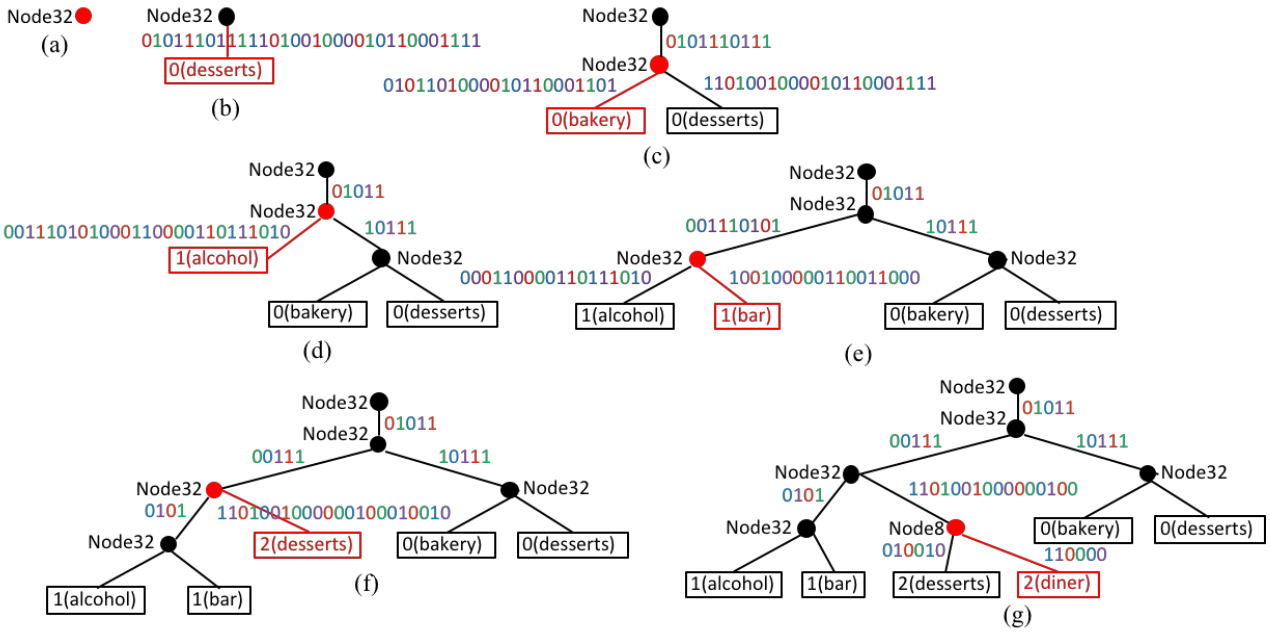

Figure 3: Progression of inserting the six keys of the three documents shown in Table 2 into a STILT$_{stx}$ index with $L = 32$. Red nodes and edges indicate where each new key is inserted. (a) Initial empty trie. (b) After inserting the first keyword from document 0. (c) After inserting the second keyword from document 0. Both keywords share the first 10 bits, introducing a new `Node32` to store the path to each key. (d) The first keyword of document 1 introduces a new `Node32` at depth 1 as the existing depth 1 `Node32` from step (c) shares only the first five bits. (e) The second keyword of document 1 has the same first 14 bits as document 1's first keyword. (f) Keyword 1 of document 2 is inserted. (g) A new `Node8` is created as both the left and right paths to the keys of document 2 require less than 8 bits each. Note that unchanged long edge paths already shown in a previous subfigure are omitted in later subfigures.

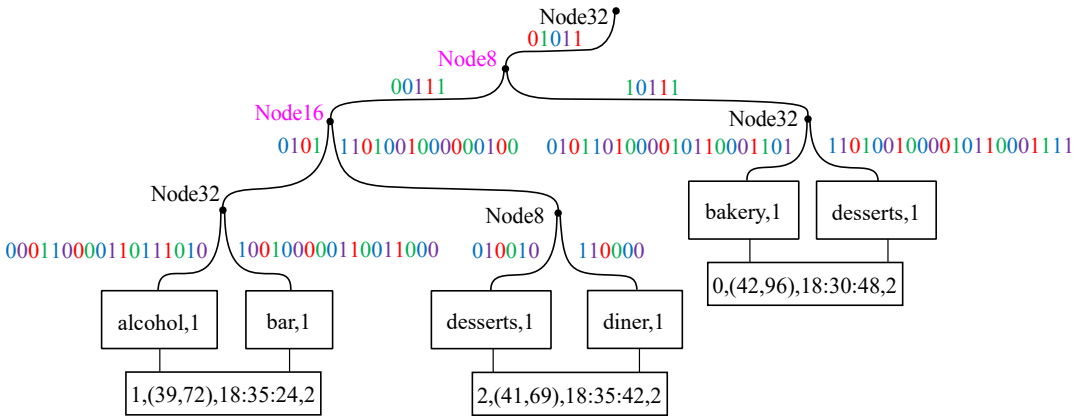

Figure 4: The final STILT index, showing all edge paths resulting from indexing the three documents in Table 2. Magenta colored nodes are compressed from `Node32` (as seen in Figure 3, step (g)). Leaf nodes reflect their content as defined in Figure 2, where the SlimKey class points to common attributes shared by all keywords in a document.

Table 3: Bytes of Key object memory usage in Java. Total size includes the JVM 12 byte header, and padding to an 8 byte boundary.

Type	int	float	long	ref	total size
FlatKey	3	2	1	1	48
SlimKey	1	0	0	2	24
SlimCommon	2	2	1	0	40

Algorithm 5: Range search using STILT.

Input : A persistent storage LSM, a root node root, and a query query.
Output : All the documents matching the search criteria.

```
1  Function range_search_stilt(LSM, root, query) : List
2      entries ← ∅
3      foreach w in W do
4          entries ← entries ∪ search_node(root, query, 0,
             FULL_RANGE)
5      docs ← ∅
6      foreach unique id in entries do
         docs ← docs ∪ get(LSM, id)
7      return docs
```

one word in W. Second, for each mapping $m = ((\ell', w', t') \rightarrow id)$ in each N, we add id to a result set R if ℓ' is within the rectangle defined by $\mathcal{L}$, w' is in W, and t' is in $\mathcal{T}$. Finally, we retrieve all documents from the persistent store whose id is in R; the collection of these documents is the result. In our implementation, we perform the first and second steps simultaneously (line 4) to reduce memory usage, which we describe in Algorithm 6. The process of retrieving data from a LSM-tree in the third step is explained in [20].

The steps to search for query-matching leaf nodes is defined in Algorithm 6. It defines two functions (lines 1 and 30), which are collectively recursive (they call each other). The functions carry a 4-dimensional search range with them, which shrinks as they approach leaf nodes; this range determines if a node or its children may be within the search criteria. The `search_node` function has two main behaviors. If it is passed a leaf node, it gathers all relevant ids (line 27) contained in the leaf and returns them (line 29). If it is passed a non-leaf node, it determines which edges are worth searching (lines 13 and 18) and initiates `search_edge` with the modified range as appropriate. The `search_edge` function processes the edge's path (line 32) to determine range at the node. If the node is within the range (line 39), `search_node` is initiated with the modified range (line 41).

5.4 Top-k Search

Algorithm 7 describes spatio-temporal textual top-k search. It takes the same parameters as range search, but adds an integer parameter k (*Definition 3.2*).

Line 2 finds all documents matching any of the query keywords in W. Lines 4 through 6 compute the spatial score $S_s(q, o)$ (Equation 3), temporal score $S_t(q, o)$ (Equation 4), and the term frequency part of the word (textual) score (Equation 6). The spatial score $S_s(q, o)$ is based on the distance $d(q.\ell, o.\ell)$ from the query rectangle center location $q.\ell = (\mathcal{L}.y_L + (\mathcal{L}.y_U - \mathcal{L}.y_L)/2, (\mathcal{L}.x_L + (\mathcal{L}.x_U - \mathcal{L}.x_L)/2)$ to the document location $o.\ell$. The term R in (Equation 3)

Algorithm 6: Search for query-matching leaf nodes.

```
1  Function search_node (node, query, depth, range) : List
2      if depth < 64 then not a leaf
3          dimension ← depth mod 4
4          depth ← depth + 1
5          synchronized ( node )
6              edge_l ← node.leftEdge
7              edge_r ← node.rightEdge
8          v ← get(range, dimension)
9          if v is * then dimension is ignored
10             l ← search_edge(edge_l, query, depth, range)
11             u ← search_edge(edge_r, query, depth, range)
12         else dimension is not ignored
13             if query intersects lower_half(v) then
14                 range′ ← clone(range)
15                 set(range′, dimension, lower_half(v))
16                 l ← search_edge(edge_l, query, depth,
                     range′)
17             else l ← ∅
18             if query intersects upper_half(v) then
19                 range′ ← clone(range)
20                 set(range′, dimension, upper_half(v))
21                 u ← search_edge(edge_r, query, depth,
                     range′)
22             else u ← ∅
23         return l ∪ u
24     else is a leaf
25         entries ← ∅
26         foreach entry in node.entries do
27             if entry.key matches query then
28                 entries ← entries ∪ {entry}
29         return entries
30 Function search_edge (edge, query, depth, range) : List
31     path ← edge.path
32     foreach i in [0 .. edge.length) do adjust range
33         dimension ← (depth + i) mod 4
34         v ← get(range, dimension)
35         if v is not * then
36             if msb(path << i) then v ← upper_half(v)
37             else v ← lower_half(v)
38             set(range, dimension, v)
39     if range matches query then
40         depth ← depth + edge.length
41         search_node(edge.node, query, depth, range)
```

normalizes the spatial score to be in the range [0,1]. The temporal score $S_t(q, o)$ is based on recency of the document time $o.t$ within the query time range (Equation 4). The final word score $S_w(q, o)$ using cosine similarity (Equation 5) is combined with the spatial and textual scores using weights (Equation 7) at line 9.

Algorithm 7: Top-K search.

Output: A list of at most k ($document \rightarrow score$) mappings.

```
1  Function topk_search(LSM, root, query, k) : List
2      entries ← range_search_stilt(LSM, root,
         (∗, query.keywords, ∗))
3      scoring_map ← new IncrementalScoringMap(query)
4      foreach entry in entries do
5          scorer = scoring_map.getsert_scorer_for(entry)
6          scorer.accept(entry.word, entry.frequency)
7      results ← new BoundedPriorityQueue(k)
8      foreach (id, scorer) in scoring_map do
9          results.add((id → scorer.finalize_score(query)))
10     return results
```

$$S_s(q,o) = 1 - \frac{d(q.\ell, o.\ell)}{R} \tag{3}$$

$$S_t(q,o) = 1 - \frac{o.t - \mathcal{T}_L}{\mathcal{T}_U - \mathcal{T}_L} \tag{4}$$

$$S_w(q,o) = \frac{\sum_{w \in q.W} \text{tf-idf}(w,o)\text{tf-idf}(w,q)}{\sqrt{\sum_{w \in q.W} \text{tf-idf}(w,o)^2}\sqrt{\sum_{w \in q.W} \text{tf-idf}(w,q)^2}} \tag{5}$$

$$\text{tf-idf}(\tau, p) = \frac{|\tau \cap p.W|}{|p.W|} * \log \frac{|O|}{|\forall o \in O : \tau \in o.W|} \tag{6}$$

$$\begin{aligned} &\textbf{Given } \text{weights } \alpha, \beta, \gamma \in [0 \,..\, 1] : \alpha + \beta + \gamma = 1 \\ &S(q,o) = \alpha S_s(q,o) + \beta S_t(q,o) + \gamma S_w(q,o) \end{aligned} \tag{7}$$

6 Analysis

As described in Table 1, the maximum height L of the binary Patricia trie defining a STILT index is either 63 or 64. Section 4 describes the STILT index structure. Interior nodes store the common prefixes of interleaved bits of the unique words in the set O of documents being indexed, and require up to 24 bytes each. Leaf nodes store the document keys $o = (id, \ell, w, t)$, where id is the document identifier, $\ell = (y, x)$ is its spatial location, $w \in W$ is one of the set W of unique words for the document, and t is the document's timestamp. With deduplication (see section 5.2), leaf nodes require a total of $40mn + 24m$ bytes, where n is the number of documents stored in the index, and m is the average number of unique words in each document (column 3 in Table 4). Keys with the same binary key (of length L) all reside in the same leaf bucket. The space required by a STILT index is therefore dominated by the leaf nodes, leading to the following theorem:

Theorem 6.1. *The main memory storage cost $S(n,m)$ of a STILT index is $\Theta(mn)$, where $n = |O|$, and m is the average number of unique words per document in the set O of documents stored in the index.*

Lemma 6.2. *Under the pointer machine model [1], the expected cost $I(o)$ to insert a single document o into a STILT index is $\Theta(m)$.*

[1] https://en.wikipedia.org/wiki/Pointer_machine

Proof. The cost $I(o)$ to insert a single document o into a STILT index is the cost of inserting $|o.W|$ binary keys, where $o.W$ is the set of unique words in document o. As shown in Algorithm 3, inserting a single binary key B requires the traversal of existing nodes until a non-matching bit or a leaf node is encountered. A non-matching bit in B causes a new interior node to be created (line 14), costing $O(1)$ time. Encountering a leaf node results in the key's attributes being inserted into a new or existing leaf node, as described in Figure 2. In both of these cases, the cost of insertion is $O(1)$ time under the pointer machine model. In the worst case, L interior nodes are traversed on the way to a new or existing leaf node, costing L time. Here, L is a constant, so the cost to insert a single binary key B is $O(1)$ under the pointer machine model. The expected number of unique words $|o.W|$ in a document o is m. □

The entire STILT index contains n documents, with an average of m unique words in each, which leads to the following theorem:

Theorem 6.3. *Under the pointer machine model, the expected cost to insert n documents having an average of m unique words per document into a STILT index residing in main memory is $\Theta(mn)$.*

The cost for range search in a main memory STILT index is determined by Algorithms 5 and 6. Under the pointer machine model, the cost to perform a range search on a STILT index is the number of nodes visited during the search. We assume the n documents $o = (id, \ell, W, t)$ are each defined by a 3-dimensional point (ℓ_y, ℓ_x, t) distributed in a uniform random fashion in space.

Theorem 6.4. *Given a STILT$_{stx}$ index built from a set of n documents having an average of m unique words per document defined by points uniformly distributed on $[0,1]^3$, and with a 3-dimensional query hyperrectangle with centre $z \in [0,1]^3$ having side lengths $\{\Delta_y, \Delta_x, \Delta_t\} \in [0,1]$, the expected cost to perform a range search is $O(\frac{m}{a}\ |q.W|\ n\ \Delta_y\Delta_x\Delta_t\ )$, where a is the number of text buckets and $|q.W|$ is the number of words in the query.*

Proof. We adapt Theorem 3.4 of [19] to range search as defined in Definition 3.1. The query components $q.\mathcal{L}$ and $q.\mathcal{T}$ define a 3-dimensional query hyperrectangle Δ of side length $q.\mathcal{L}.y_U - q.\mathcal{L}.y_L$, $q.\mathcal{L}.x_U - q.\mathcal{L}.x_L$, and $q.\mathcal{T}_U - q.\mathcal{T}_L$. In addition, we search for a set of query keywords $q.W$ matching document keywords stored in the STILT$_{\text{stx}}$ index. Without loss of generality, we normalize the three side lengths by dividing by the size of the space being indexed in each dimension. As range search proceeds down the binary Patricia trie (see Algorithm 6), an edge is followed only if the interior node N's hyperrectangle intersects Δ, and if the text bits found when following edges up to N match the query keyword w being searched for (line 3 in Algorithm 5). The normalized volume of the indexed space $[0,1]^3$ visited is $\Delta_y\Delta_x\Delta_t$, and $n\Delta_y\Delta_x\Delta_t$ defines the fraction of the 3-dimensional space visited. Assuming equal probability of words falling into one of a text buckets in a STILT$_{\text{stx}}$ index containing n documents, the expected number of documents containing a given word is defined by the following:

$$\text{recurrence: } f(1) = 1/a, f(n) = a^{-n} + f(n-1) \tag{8}$$

$$\text{solution: } f(n) = \frac{1 - a^{-n}}{a - 1} \tag{9}$$

The second term $a^{-n}/(a-1)$ of the solution in Equation 9 can be ignored as as it is extremely small compared to the first term

Table 4: Datasets. Keys and words per document.

Dataset name	Num. of docs.	Mean keys	Max keys	Mean words
Spaten	1,523,849	21.75	62	23.26
Tw20mi	20,000,000	5.48	38	5.70
Wikipedia	280,000	390.14	7,770	1,018.39

Table 5: Parameter setting.

Parameter	Setting
Datasets	Tw20mi, Spaten, **Wikipedia**
Number of queries	**1000**
k	**5**, 10, 15, 20, 25
Num. query keywords	1, 2, 3, 4, **5**
Number of threads	1, 2, 4, 8, **16**

$1/(a-1)$. Each document contains an average of m unique words, so Algorithm 6 visits up to m buckets for each document intersecting the 3-dimensional query hyperrectangle Δ. □

The number a of text buckets is determined by how many bits are used to represent characters in the alphabet representing words in the index. In this paper, Algorithm 1 is used to map each character in a word to 5 bits, with $b = 16$ bits in total used for indexing text in a $\text{STILT}_{\text{stx}}$ index, providing buckets for up to 2^{16} text prefixes. Five bits are used to represent each character, giving three complete characters, and one bit of the fourth character as a prefix. For example, all words starting with letters "che" followed by characters with a first bit of "0" will fall into the bucket with text bits `00100 01001 00110 0`. The number of English words in common use is approximately 170,000 [29]. Many English words have common prefixes, resulting in fewer than 2^{16} text buckets.

For top-k search, Algorithm 7 visits many more nodes due to the cost of visiting all nodes in the 3-dimensional indexed space.

7 Evaluation

The following sections describe our experimental setup, datasets, query sets, and performance evaluation involving index construction and query execution. The parameter settings are shown in Table 5, with the default parameters in bold font.

7.1 Experimental setup

7.1.1 Datasets We used three real world datasets, Twitter (Tw20mi), Wikipedia [28], and Spaten [7]; the details can be found in Table 4. The documents in all datasets include geo-location and timestamp information.

7.1.2 Query sets Our query sets consist of 1,000 queries. The query set was formed by randomly selecting documents from the datasets in 25% of the queries, forcing at least 25% of searches to return non-empty results. The rest of the queries consisted of random values generated from a global dictionary, which was formed by combining tokens from the datasets. Each set of 1,000 queries was run three times (after one warm-up run).

7.1.3 Environment setup The experiments were conducted on a machine with 16 AMD Opteron cores and 128 GB RAM. All our code is written in Java 9 and we ran our experiments on on Ubuntu 16.04 OS. We implemented our own version of ST2I based on its definition in [8]. We obtained B^{x}-tree [9] and I^3 [30] from their respective authors. For the persistent storage, which is based on the LSM-tree, we used LevelDB's implementation.

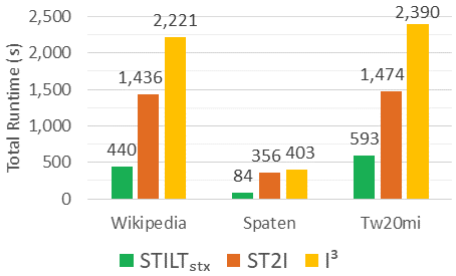

Figure 5: Index construction time.

Table 6: Query experiments setting.

Category	Query	STILT variants	Compared against
Spatio-temporal textual	Top-k	$\text{STILT}_{\text{stx}}$	ST2I
	Range	$\text{STILT}_{\text{stx}}$	ST2I
Spatio-textual	Top-k	STILT_{sx}	I^3
Spatio-temporal	Range	STILT_{st}	B^{x}-tree

7.2 Index construction

Although STILT does support simultaneous insertion and search, we separate our experiments into a build (index construction) and a query phase. This is because none of our comparison indexes support simultaneous data insertion and query execution.

7.2.1 Index construction cost Figure 5 shows the index building time for STILT, ST2I and I^3, for all three datasets. STILT is by far the fastest, being up to 3.3× faster than ST2I and 5.0× faster than I^3. We attribute STILT's speed advantage to the fact that it resides entirely in memory, combined with its ability to be built progressively. Together, these features enable significant scaling with the number of processors available. By contrast, both ST2I and I^3 have components written to disk, and they both need to be built all at once. This means that while STILT can be built as the data is being read, the other two indexes need to load the entire dataset in memory before they can begin indexing. We omitted the B^{x}-tree from this figure because it took more than two hours to construct.

7.2.2 Index storage usage Figure 6 shows the storage usage of STILT, ST2I, and I^3 during index construction. The lower bars (dark) show the Java-reported heap size, while the lighter bars show disk usage. Note that STILT only has a single dark bar, because it does not use disk storage. I^3 is competitive for Spaten, but is significantly outclassed for Tw20mi and Wikipedia. STILT and ST2I are competitive with each other's total memory usage for all three datasets. While it is true that ST2I requires less main memory than STILT, it uses a memory-mapped file for its disk storage. This means that to perform optimally, some or all of its disk storage will be duplicated in main memory, making its main memory usage comparable to STILT's. This is the case in our experiments since our machine has abundant memory for the datasets used here.

7.3 Query performance

We evaluated the performance of both top-k search and range search with the indexes. A summary of the query experiments setting is

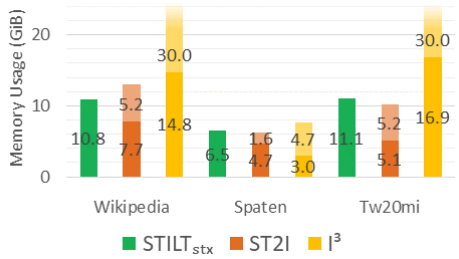

Figure 6: Index storage usage (I^{3}extends beyond y axis).

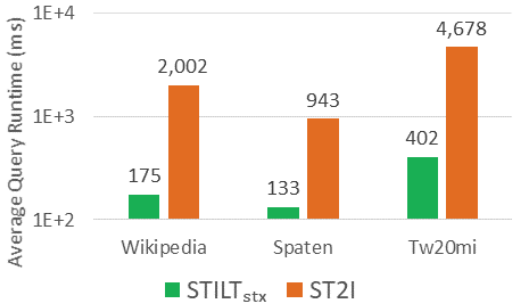

Figure 7: Top-k search (spatio-temporal textual): vary datasets.

shown in Table 6. Except for experiments involving Figures 7 and 8 (where the datasets were varied), all experiments mentioned in the following sections were conducted with the dataset Wikipedia. Except for experimental results shown in Figures 9 and 10, all experiments are performed using spatio-temporal textual queries. As reported for ST2I [8], our average query runtimes include the time to search the index, and return the unique document *id*s, but not the time to retrieve documents from persistent storage.

7.3.1 Vary datasets We compare STILT$_{stx}$ with ST2I in spatio-temporal-textual top-k (Figure 7) and range (Figure 8) search.

STILT$_{stx}$ performs up to 11.6× faster than ST2I in top-k search and up to 18.2× faster in range search, both with the Tw20mi dataset. STILT's lead is due to its efficient pruning and the fact that its index structure is stored in memory, allowing parallel index look-ups with virtually no contention between threads. All other indexes are single-threaded and rely on some form of disk storage for their index, which significantly slows them down.

7.3.2 Subset dimensional search We compare STILT$_{st}$ against B^x-tree for spatio-temporal range search (Figure 9), and STILT$_{sx}$ with I^3 for spatio-textual top-k search (Figure 10). STILT$_{st}$ performs four orders of magnitude faster than B^x-tree, while STILT$_{sx}$ is one order of magnitude faster than I^3. Note that the B^x-tree was designed as a spatio-temporal index for moving objects, whereas the documents in our datasets are static. Hence, we adapted our dataset to work with B^x-tree by setting the velocity of objects to zero and generating unique timestamps, which were compatible with B^x-tree.

7.3.3 Top-k search: vary k We evaluate top-k search by varying k (Figure 11). Neither index is significantly affected by varying k. STILT$_{stx}$ is an average of 11.6× faster than ST2I.

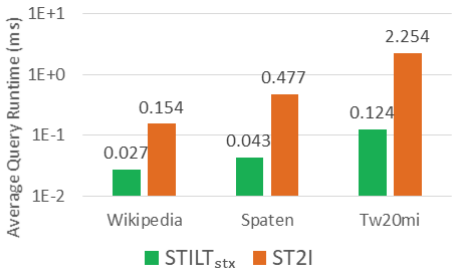

Figure 8: Range search (spatio-temporal textual): vary datasets.

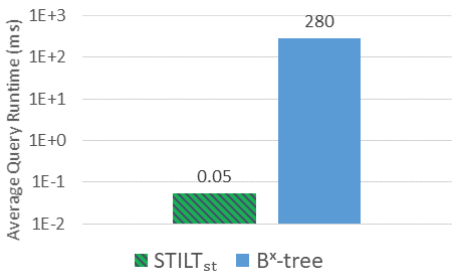

Figure 9: Range search (spatio-temporal).

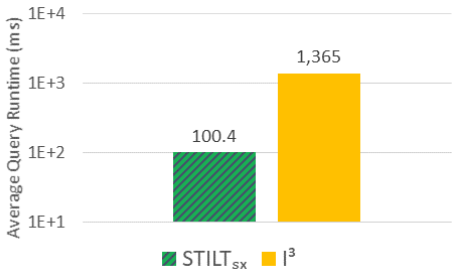

Figure 10: Top-k search (spatio-textual).

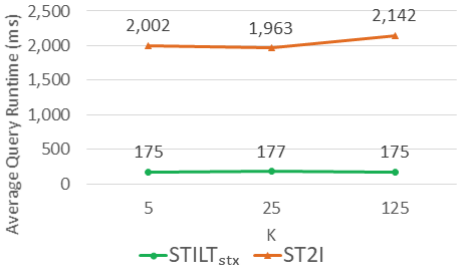

Figure 11: Top-k search (spatio-temporal textual): vary k.

7.3.4 Top-k search: vary number of keywords For these experiments, we vary the number of query keywords as 1, 3, and 5 (Figure 12). This variable is a good predictor for performance as both STILT$_{stx}$ and ST2I scale similarly, as expected since they both process keywords sequentially. STILT$_{stx}$ is 8.9× faster than ST2I on average.

7.3.5 Top-k and range search: vary number of threads We evaluate STILT's ability to scale across processors by varying the

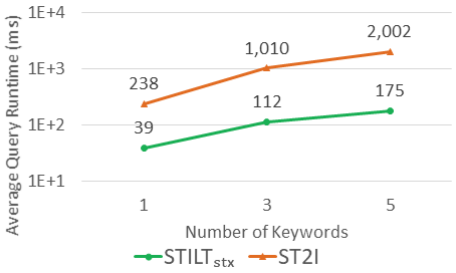

Figure 12: Top-k search (spatio-temporal textual): vary number of keywords.

number of threads (Figure 13). Scaling is nearly linear from 1 to 8 threads with a parallel efficiency of 0.96. The scaling drops slightly as we saturate our machine's processors with 16 threads, achieving a parallel efficiency of 0.82.

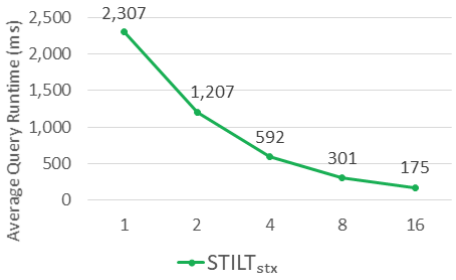

Figure 13: Top-k search (spatio-temporal textual): vary threads

7.4 Discussion

STILT is very efficient for both range and top-k search due to its novel use of a binary PATRICIA trie to discretize the space being indexed. The STILT trie is similar to a quadtree in its partitioning of 2-dimensional space, but is much smaller in size due to its removal of one-child nodes, unified space partitioning across all four dimensions, and adaptive node size. With its multi-threading capability, STILT is able to achieve an average of an order of magnitude speedup in both top-k and range search compared to other state-of-the-art indexing techniques. Index construction time is between 2× and 5× faster for all three datasets compared to ST2I and I^3. Speed of construction is enhanced by STILT's static partitioning of space, which is ideally suited for multi-threaded insertion. The storage usage for a STILT$_{stx}$ index is very comparable to ST2I, and about 4× smaller than an I^3 index.

8 Conclusion

With the continuous growth of data that includes text, time, and location components, it is important to develop efficient indexes to support spatial, temporal and textual search. We introduced a generalized multi-dimensional index called STILT, which supports efficient spatio-temporal textual range search and top-k search using a combined ranking scheme. STILT also supports subset dimensional searches (e.g. spatio-textual, spatio-temporal, spatial, textual) in a unified manner. STILT is a multi-dimensional binary trie index suitable for modern multi-core machines that support multi-threading, and that uses an adaptive node structure to achieve memory efficiency. With extensive experimental evaluation, we demonstrated that STILT significantly outperforms other state-of-the-art indexes.

Acknowledgments

This research was supported in part by Natural Sciences and Engineering Research (NSERC) Discovery Grants 36866-2011-RGPIN (Nickerson) and RGPIN-2016-03787 (Ray) and NBIF Start-Up Grant (Ray).

References

[1] Abdulaziz Almaslukh and Amr Magdy. 2018. Evaluating Spatial-keyword Queries on Streaming Data. In *ACM SIGSPATIAL*. 209–218.
[2] Robert Binna et al. 2018. HOT: A Height Optimized Trie Index for Main-Memory Database Systems. In *SIGMOD*.
[3] Su Chen et al. 2008. ST2B-tree: A Self-tunable Spatio-temporal B+-tree Index for Moving Objects. In *SIGMOD*.
[4] Su Chen, Christian S. Jensen, and Dan Lin. 2008. A Benchmark for Evaluating Moving Object Indexes. *Proc. VLDB Endow.* 1, 2 (Aug. 2008), 1574–1585.
[5] Maria Christoforaki, Jinru He, C. Dimopoulos, Alexander Markowetz, and Torsten Suel. 2011. Text vs. Space: Efficient Geo-search Query Processing. In *CIKM*.
[6] Gao Cong, Christian S. Jensen, and Dingming Wu. 2009. Efficient Retrieval of the Top-k Most Relevant Spatial Web Objects. *Proc. VLDB Endow.* (2009).
[7] Thaleia Dimitra Doudali, Ioannis Konstantinou, and Nectarios Koziris. 2017. Spaten: A spatio-temporal and textual big data generator. In *IEEE BigData*.
[8] Tuan-Anh Hoang-Vu, Huy T. Vo, and Juliana Freire. 2016. A Unified Index for Spatio-Temporal Keyword Queries. In *CIKM*.
[9] Christian S. Jensen, Dan Lin, and Beng Chin Ooi. 2004. Query and update efficient B+-tree based indexing of moving objects. In *VLDB*. 768–779.
[10] Ali Khodaei, Cyrus Shahabi, and Chen Li. 2010. Hybrid Indexing and Seamless Ranking of Spatial and Textual Features of Web Documents. In *DEXA*.
[11] Peter Kirschenhofer and Helmut Prodinger. 1994. Multidimensional Digital Searching-Alternative Data Structures. *Random Struct. Algorithms* 5, 1 (1994).
[12] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The Adaptive Radix Tree: ARTful Indexing for Main-memory Databases. In *ICDE*.
[13] Z. Li, K. C. K. Lee, B. Zheng, W. Lee, D. Lee, and X. Wang. 2011. IR-Tree: An Efficient Index for Geographic Document Search. *TKDE* (2011).
[14] Dan Lin et al. 2005. Efficient indexing of the historical, present, and future positions of moving objects. In *MDM*.
[15] Amr Magdy et al. 2014. Taghreed: A System for Querying, Analyzing, and Visualizing Geotagged Microblogs. In *ACM SIGSPATIAL*. 163–172.
[16] Amr Magdy, Mohamed F. Mokbel, Sameh Elnikety, Suman Nath, and Yuxiong He. 2014. Mercury: A memory-constrained spatio-temporal real-time search on microblogs. In *ICDE*.
[17] Ahmed R. Mahmood et al. 2015. Tornado: A Distributed Spatio-textual Stream Processing System. *Proc. VLDB Endow.* 8, 12 (2015), 2020–2023.
[18] Donald R. Morrison. 1968. PATRICIA–Practical Algorithm To Retrieve Information Coded in Alphanumeric. *J. ACM* (1968).
[19] Bradford G. Nickerson and Qingxiu Shi. 2008. On k-d Range Search with Patricia Tries. *SIAM J. Comput.* (2008).
[20] O'Neil, P. and others. 1996. The Log-structured Merge-tree (LSM-tree). *Acta Inf.* (1996).
[21] Dieter Pfoser, Christian S. Jensen, and Yannis Theodoridis. 2000. Novel Approaches in Query Processing for Moving Object Trajectories. In *VLDB*.
[22] Suprio Ray and Bradford G. Nickerson. 2016. Dynamically ranked top-k spatial keyword search. In *ACM SIGMOD Workshop on Managing and Mining Enriched Geo-Spatial Data (GeoRich)*. 6:1–6:6.
[23] João B. Rocha-Junior, Orestis Gkorgkas, Simon Jonassen, and Kjetil Nørvåg. 2011. Efficient Processing of Top-k Spatial Keyword Queries. In *SSTD*.
[24] String interning 2019. String interning. https://en.wikipedia.org/wiki/String_interning.
[25] Yufei Tao and Dimitris Papadias. 2001. MV3R-Tree: A Spatio-Temporal Access Method for Timestamp and Interval Queries. In *VLDB*.
[26] Subodh Vaid, Christopher B. Jones, Hideo Joho, and Mark Sanderson. 2005. Spatio-textual Indexing for Geographical Search on the Web. In *SSTD*.
[27] Xiang Wang, Ying Zhang, Wenjie Zhang, Xuemin Lin, and Wei Wang. 2015. AP-Tree: efficiently support location-aware Publish/Subscribe. *VLDBJ* (2015).
[28] Wikipedia Dump 2018. https://dumps.wikimedia.org/enwiki/latest/.
[29] Word Count 2020. List of dictionaries by number of words. https://en.wikipedia.org/wiki/List_of_dictionaries_by_number_of_words.
[30] D. Zhang et al. 2013. Scalable Top-k Spatial Keyword Search. In *EDBT*.