A Parallel Integrated Index for Spatio-temporal Textual Search Using Tries

by

Yoann S. M. Arseneau

Bachelor of Computer Science, UNB, 2019

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Computer Science

In the Graduate Academic Unit of your GAU

Supervisor(s):	Bradford G. Nickerson, Ph.D. Computer Science
	Suprio Ray, Ph.D. Computer Science
Examining Board:	Patricia Evans, Ph.D., Computer Science, Chair
	Virendra Bhavsar, Ph.D., Computer Science
	David Coleman, Ph.D., Geodesy and Geomatics Engineering, UNB

This thesis is accepted

Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

May, 2019

Abstract

The proliferation of location-enabled devices and the increasing use of 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-textual data processing either use separate spatial and textual indices, or a combined index that integrates an inverted index with a tree data structure, such as an R-tree or Quadtree. These approaches, however, do not integrate temporal, spatial, and textual data together. We propose a novel integrated index called Spatio-temporal Textual Interleaved Trie (STILT), which unifies spatial, textual, and temporal components within a single structure. STILT is a multi-dimensional binary-trie-based index that interleaves text, location, and time data in a space-efficient manner. It supports dynamic and parallel indexing as well as concurrent searching. With extensive evaluation we demonstrate that STILT is significantly faster than the state-of-the-art approaches in terms of index construction time and query latency.

Dedication

I dedicate this thesis to my parents, Gérène and Michel, who supported me emotionally and financially throughout this endeavor. Despite their early passing, their support can be felt to this day.

Acknowledgements

I wish to acknowledge my partner Jennifer, my parents Gérène and Michel, my brother Sacha, and my friends and extended family for helping me arrive at this point in my life.

I also wish to acknowledge my supervisors Bradford G. Nickerson and Suprio Ray, my colleague Saransh, and my other colleagues in the Big Data lab for their support and camaraderie throughout this venture. I also thank the examining board members for their feedback.

There are many other people who contributed to my success, including but not limited to faculty and staff at UNB Fredericton and Saint John, and teachers and staff at Samuel-de-Champlain.

Table of Contents

Vita

List of Tables

List of Figures

List of Symbols, Nomenclature, and Abbreviation

$\varnothing$	the empty set
α, β, γ	weights used to combine different scores
ART	**A**daptive **R**adix **T**rie
b	the number of bits for each of the four dimensions
c	the maximum number of iterations for a top-k query
$\texttt{clz}(i)$	the number of leading zeroes for unsigned integer i
$d(\ell_0, \ell_1)$	the distance between spatial coordinates ℓ_0 and ℓ_1
f	one of N mapping functions used by STILT
HOT	**H**eight **O**ptimized **T**rie
id	a unique identifier for a given document o
k	the maximum number of results for a top-k query
ℓ	a spatial coordinate
L	the maximum depth and exact length of STILT's trie
LBS	**L**ocation-**B**ased **S**ervice

LSM-tree	**L**og-**S**tructured **M**erge **tree**
MBR	**M**inimum **B**ounding **R**ectangle
`msb`(i)	the most significant bit for an unsigned integer i
N	the number of dimensions used for STILT
o	an arbitrary document from the set O
O	the set of all documents
p	a document or query
path	a binary string representing left(0) and right(1) decisions required to move from a parent node to one of its children
q	a range or top-k query
r	the radius for a disk centered around ℓ
$S(q, o)$	the scoring function used for top-k queries
STILT	**S**patio-temporal **T**extual **I**nter**l**eaved **T**rie
SFC	**S**pace **F**illing **C**urve
τ	a term (word)
t	a particular point in time
$t_{\text{lo}}, t_{\text{hi}}$	lower and upper bounds of a range of time, respectively
tf-idf	**t**erm **f**requency-**i**nverse **d**ocument **f**requency
trampoline	alternate name for non-concurrent continuations
w	a word, usually an element of W if defined in the context
W	a list or set of words for documents and queries, respectively
x	an arbitrary document from a result set X
X	the result of a query on the set of documents O

Chapter 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, sensors, web and social media, internet-of-things communication, enterprise applications, digital archives, and public records. A recent estimate [13] indicates that there are more than 727 million tweets per day, or 8,400 per second. According to GSMA's real-time tracker [9], the number of mobile devices world-wide is 8.95 billion. A growing number of these mobile devices are geolocation-enabled, producing massive streams of valuable geo-tagged data. 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. Leetaru [17] reports on the Twitter 1% stream from Jan. 1, 2012 through Oct. 31, 2018. His analysis shows that a change in Twitter policy in late April, 2015 resulted in a de-

crease from around 2% of geotagged tweets having precise GPS coordinates to around 0.5%. After April, 2015, the primary source for geographical location of the 1.5% of tweets that are geotagged was the centroid location of geographical place names. Sloan and Morgan [25] report that in a selected set of 30 million *users* in April 2015, 3.1% have geotagged tweets. Profile based location information may also be available for a Twitter feed, when the tweets are not explicitly geo-tagged. Another social media example are reviews about visited business venues (e.g. restaurants, bars, and shopping centers) on sites like Yelp. Blog posts and articles posted online are also a rich source 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. For time components, all documents have a few inherent data points: the time they were first created, first published, or last edited. In the event that none of those data points are known, we can simply choose the time at which the document was retrieved.

1.1 Motivation

These datasets involving text, time, and spatial information can provide valuable insights. 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 kd-trees 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 TPR*-tree [32], BB^x-index [20], ST2B-tree [5], STR-tree [28] and MV3R-tree [31]. Many of these systems adopt a versioned data structure approach, in which a "snapshot" of a tree or grid data structure is taken at regular time intervals. Then, while executing a search with spatio-temporal range queries, those versions of the data structure that match the query time range are processed. 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 [19] and the DIR-tree [7] 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. Additional optimization provided by S2I [30] and I^3 [38] were introduced to take advantage of the fact that in a typical text dataset, some keywords appear far more frequently than others.

As more and more data sets include text, time, and location components in the same document, it is increasingly important to support queries that specify search criteria based on all 3 components. An example query is *"Re-*

trieve 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". These queries can be processed by employing two or three different existing indexing approaches at once. 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 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 can therefore perform poorly if there is little overlap between the results of the different indices. The research community did not pay much attention to this problem until the introduction of the ST2I [12], which to our knowledge is the first index to provide a unified index and integrated ranking scheme that handles space, time, and text in a single data structure; however, it has a few limitations. First, it supports a limited character set and only supports strings up to 12 characters in length. This enables efficient bijective mapping between words and integers, but severely limits the languages that can be indexed, since it only supports alpha-numeric Latin characters. ST2I also lacks support for parallel operations to take advantage of modern multi-core processor, and it needs be entirely rebuilt every time data is added or removed from the index. These limitations limit ST2I's practicality to static environments

where new data arrives slowly and characters sets are predictable, such as historical archives. 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 [21] supports a few types of queries such as spatio-temporal boolean range keyword search and top-k frequent keyword query; however, it does not support a ranking scheme that considers both spatial and textual relevance. Mercury [22] is a pyramid-based in-memory index designed to support top-k spatio-temporal textual queries. Its ranking mechanism, however, only incorporates spatial and temporal relevance; textual relevance is not considered.

To address the issues with existing approaches, we introduce an integrated spatio-temporal textual index that we call Spatio-temporal Textual Interleaved Trie (STILT). It is a multi-dimensional binary Patricia trie [24]. Tries are highly efficient on modern hardware for in-memory indexes. It has been noted by researchers [2] that well-designed tries can outperform comparison based indexes (such as B-tree or R-tree and their variants), particularly with string data. STILT converts indexable components (e.g. spatial, textual, temporal) to positive integers via a linear mapping. These integers are then bit-interleaved as follows:

$$y_1 x_1 w_1 t_1 y_2 x_2 w_2 t_2 \ldots y_b x_b w_b t_b \tag{1.1}$$

where y_1, x_1, w_1, t_1 refer to the first bit of the latitude, longitude, word, and time, respectively; b is the number of bits for each of the four dimensions. Figure 3.2 shows an example STILT index with three documents (shown in Table 3.1 with $b = 8$).

STILT was specifically designed to handle limitations of previous implementations. To handle arbitrarily long words, we index them based on their prefix. To handle character set mismatching, our textual encoding reserves a special value to represent characters not otherwise defined; this means that unexpected characters in documents will be properly indexed and searchable in the main data structure. To handle the rise of continuous data streams, we designed STILT to handle iterative and parallel insertion, as well as to support searching while insertions are in progress — it never needs to be taken offline to be rebuilt or queried.

To evaluate STILT, we conducted a comprehensive experimental study with 3 real-world datasets (see Table 5.1), against the state-of-the-art integrated index, ST2I. In fact, we extended ST2I to support parallel index construction and concurrent query execution, which are not supported by the original ST2I as described in [12]. Experimental results suggest that STILT 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 4.8× faster than that of ST2I. Furthermore, the range search execution time of STILT is 6.3× (Wikipedia dataset) and top-k search execution time is 2× (Twitter 20mi dataset) faster than those

of ST2I. We also evaluated STILT against a modified spatio-textual index I^3 index [38], which is considered to be one of the fastest spatio-textual indexes; we added support for a temporal component to its preexisting top-k query. With the Wikipedia dataset, STILT was 6.4× and 22.6× faster than I^3 in index construction and top-k search times, respectively. The modified I^3 index performed significantly worse than both STILT and ST2I in all cases.

1.2 Contributions

The principle contributions of this thesis are as follows:

- We developed a novel, space efficient, spatio-temporal textual index STILT, based on a multi-dimensional binary trie, which uses an adaptive node allocation strategy.
- Our system supports parallel index building and concurrent query execution, exploiting modern multi-core machines.
- The STILT index resides entirely in memory to provide efficient search.
- In our experimental evaluation involving 3 real-words datasets, STILT performed significantly better than ST2I and an extended I^3 index.
- On the datasets we used, STILT can be built faster than ST2I and I^3.
- STILT uses no more memory to build its index than to search it

- STILT places the burden of boundaries on the programmer, but it affects distribution predictably. This fact can be exploited to increase performance.
- STILT requires less RAM than ST2I for optimal performance, when we take into consideration the size of ST2I's memory mapped file.

Chapter 2

Related work

First, we present related work on trie-based indexes. We then discuss prior research pertaining to spatio-textual, temporal-textual, and spatio-temporal indexes, respectively. Finally, we outline existing work on integrated spatio-temporal textual indexes.

Note that the objectives of our parallel integrated index include supporting concurrent spatio-temporal textual data ingestion and ranked *ad hoc (snapshot)* query processing in a modern server class machine having large memory and many cores. Therefore, distributed spatio-textual indexing approaches, such as Tornado [23], are beyond the scope of this thesis. Publish-subscribe indexing approaches for continuous spatio-temporal queries, such as TaSK [3] or AP-Tree [35], are also outside our scope.

2.1 Trie-based Index

A trie is a data structure in which each child of a node shares a common prefix. As such, the search process involves deciding which child to proceed with depending on remaining components of a search key. A binary Patricia trie [24] can reduce the tree height by skipping nodes having only one child. 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) [18] 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 Patricia trie into compound nodes with a maximum predefined fanout.

STILT is a multi-dimensional binary Patricia trie with an adaptive node-allocation strategy to save memory. ART and HOT each also employ trie-based approaches; however, they do not handle multi-dimensional data, involving spatial, temporal, and textual components like STILT.

2.2 Spatio-temporal Index

Due to the rising popularity of Location-Based Services (LBS), a number of spatio-temporal indexes were proposed. Many of these approaches utilize a versioned organization that maintains particular data structure "snapshots" obtained at regular time intervals.

The BB^x-index [20] is a technique that keeps a forest of B^x-trees [14], each tree for a different time interval. A B^x-tree exploits a B-tree to index the object locations converted into space filling curve (SFC) codes. The ST2B-tree [5] is another B-tree based technique.

Several R-tree based approaches have also been proposed. The STR-tree [28] uses an R-tree to index moving object trajectories. The MV3R-tree [31] uses multiple versions of R-trees that resemble a BB^x-index.

2.3 Spatio-textual Index

In recent years, spatial keyword search has become increasingly important. In response, a number of spatio-textual indexes have emerged which can be classified into three groups based on their structure: R-tree based, grid-based, and space-filling-curve based. Chen et al. [4] conducted an evaluation study of 12 of these indexes and reported that not all of them support the top-k spatial keyword search query.

The R-tree based approaches generally combine an R-tree with an inverted file, the IR-tree [7] being a classic example. It augments the nodes of the R-tree with summary information of the textual content of the node's corresponding subtree. In this approach, an upper bound of the textual relevance score is computed from the summary information, and the spatial relevance is calculated from the MBR. This can be used for pruning search paths. The authors also present variants of the IR-tree, namely, the DIR-tree, the CIR-

tree and the CDIR-tree. To deal with the widely varying frequency of terms, the S2I [30] index uses different approaches. For infrequent keywords, all the elements in the inverted file are stored sequentially to minimize I/O. For frequent keywords, an aggregated R-tree (aR-tree) is used for pruning. The I^3 index [38] 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 [6] is a space filling curve approach that orders documents on their position in a Z-curve and stores them in an inverted list.

The grid-based indexes normally integrate a grid index with a textual index. Among these, the two most prominent ones are ST & TS [34] and SKIF [16]. To maintain the textual information, SKIF utilizes an inverted-file structure. ST and TS are spatial-first and textual-first indices, respectively. Pastri [29] is a hybrid index that integrates a grid with an STR R-tree; it maintains inverted lists in each cell of a grid. It assumes the locations of each document as a region, rather than a point, and the cells that overlap the query window are considered for further processing.

While most spatio-textual indexes support range queries, only a few of them support top-k spatial keyword search. The R-tree based approaches, such as IR-tree, S2I and I^3 support it. SFC-QUAD [6] is among only a few non-R-tree based approaches to support top-k search. As noted by [4], the grid-based approaches only support boolean range queries, but not top-k search.

2.4 Temporal-textual Index

Although a number of indexes have been proposed that support temporal keyword queries, only a few incorporate temporal and textual relevance scores [12]. T^2I^2 [15] is a grid-based approach that combines temporal and textual elements into an inverted list. However, its storage requirement is quite high and it is susceptible to data skew. To address these issues, He [10] explored ways to partition data in order to organize documents by time.

2.5 Unified Spatio-temporal Textual Index

According to the authors [12], ST2I is the first index that integrates spatial, temporal, and textual components in a single structure. It index consists of a compact kd-tree (KTree) and the data is stored on disk in a fixed size array of `KBlock`s, called the KBlock file. The `KBlock`s store `KPoint`s, which are $(key \rightarrow id)$ mappings. Each successive level of the the KTree alternates between latitude, longitude, text, and time. The internal nodes in maintain the splitting point of their dimension along with a pointer to the left and right child nodes, whereas the leaf nodes contain an offset in the KBlock file. As mentioned in section 1.1, ST2I has several limitations including a lack of support for non-Latin character sets, the need for complete index reconstruction in to insert new data, and a lack of support for concurrent index building and query execution. ST2I uses functions `word_to_id` and `id_to_word` to map terms to number and vice versa; however, the algorithms used do not handle

non-alphanumeric characters in words, which can exist in many documents. Another issue is that the character '0' to map to the number zero, which is indistinguishable from the lack of a character. This is not a problem for zeroes which appear after other, non-zero characters; however, when a word is prefixed with zeroes, there is no way to distinguish the absence of characters (a shorter string) from a string prefixed with zeroes. For example, the string `"a"`, `"0a"`, and `"00a"` all map to the number ten. The `id_to_word` function will invariably convert the number ten to the string `"a"`, breaking the bijective property. This can be remedied by reserving a value to represent the absence of a character.

Other than ST2I, a few other indexes were designed to support queries on data with spatial, temporal, and textual attributes. These systems include Taghreed [21], Mercury [22], and STTIA [1]; however, unlike ST2I, they neither incorporate space, time, and text in a unified data structure, nor support a combined spatio-temporal textual ranking scheme.

Chapter 3

The STILT Index

In this chapter, we discuss the STILT data structure and its use as an index. We begin with an overall description of its structure and components, followed by an example. We then explain how to build, search, and compress the index. Finally, we discuss concurrency support.

3.1 Structure

The STILT index is an in-memory Patricia trie [24, 26, 2] designed to support highly efficient insertion and search. It maps multi-dimensional keys to ids and relies on an external data store to map the ids to documents. STILT requires a pre-determined maximum length $L \in \mathbb{Z}^+$, a number of dimensions $N \in \mathbb{Z}^+$, a mapping function for each dimension, and a *path schedule*. To map a key to a leaf, we map each of its keys into 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 the *path schedule*, which dictates the order in which the bits are to be concatenated to produce a path. The schedule also implicitly dictates how many significant bits are needed for each integer. We chose a balanced schedule, as shown in the following example.

$$\text{Let } y, x, w, t \text{ be the integer mappings for a key}$$
$$\text{path} = y_1 x_1 w_1 t_1 y_2 x_2 w_2 t_2 \ldots y_n x_n w_n t_n | n = \lfloor L/4 \rfloor$$

A path maps to a leaf node by interpreting its bits as left or right choices (i.e. 0 = go left, 1 = go right) as we descend from a parent node.

3.2 Components

Our unique implementation of the Patricia trie uses an adaptive node allocation strategy to minimize memory usage. We use instances of ***NonLeafNode***s and `LeafNode`s, as illustrated in Figure 3.1. The `LeafNode` stores ($key \rightarrow id$) mappings in its only attribute, named `items`. This attribute references either an array of mappings or a mapping -- a memory optimization which is hidden from the user via the `forEachEntry` visitor function. The ***NonLeafNode*** is composed of a left and right child node and has abstract methods for long-edge encoding. By default, the non-leaf node supports a length of one, represented by picking the left or right child; the implementations of ***NonLeafNode***

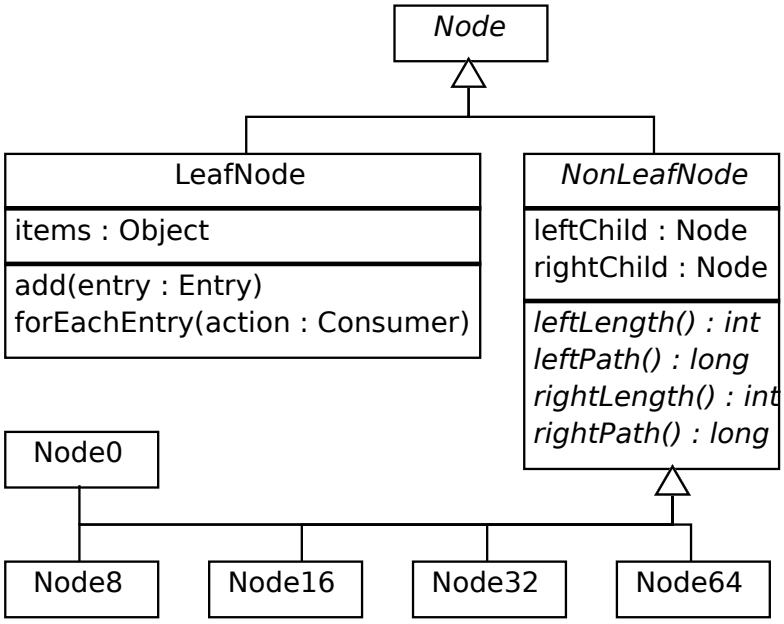

Figure 3.1: UML diagram for nodes in the STILT index.

can increase the maximum representable length of its edges by implementing the path and length methods. The length methods report how many bits of the path method are relevant, and the path represents a series of left-right choices encoded as a binary string. For example, a `Node0` can only support a path length of one because it has no attributes to store more edge information. Conversely, the `Node64` can support path lengths of up to 65; the 64 extra bits of path information are encoded in 64 bit integers. Whenever a `NonLeafNode` is created, the smallest implementation that is guaranteed to fit all possible edges is created. For example, if a STILT index has a length of 64, it is guaranteed that a non-leaf node at a length of 63 can only have immediate children; therefore, a `Node0` is sufficient. Conversely, for a

non-leaf node created at a length of 10, its children may be at a distance of up to 53; therefore, a `Node64` is required to ensure any subsequent insertion succeeds. Consequently, the root node for such a STILT index will always be a `Node64`, since the first insertion is guaranteed to result in a single edge with a path length of 64 bits.

3.3 Example

To illustrate the components, we produced the sample data set in Table 3.1 and the corresponding STILT index in Figure 3.2. The color coding indicates which parts are related to latitude, longitude, text, and time.

Table 3.1: Three example documents and their binary representation.

Doc id	Original document	Document keys	Binary representation
0	(42,96,{bakery,desserts},1539993601)	(42,96,bakery,1539993601) (42,96,desserts,1539993601)	(10010010,11111111,00010000,11111111) (10010010,11111111,00100001,11111111)
1	(39,72,{bar,alcohol},1539993601)	(39,72,bar,1539993601) (39,72,alcohol,1539993601)	(10000101,11111111,00010000,11111111) (10000101,11111111,00001011,11111111)
2	(41,69,{diner,desserts},1540191601)	(41,69,diner,1540191601) (41,69,desserts,1540191601)	(10001101,11111111,00100010,11111111) (10001101,11111111,00100001,11111111)

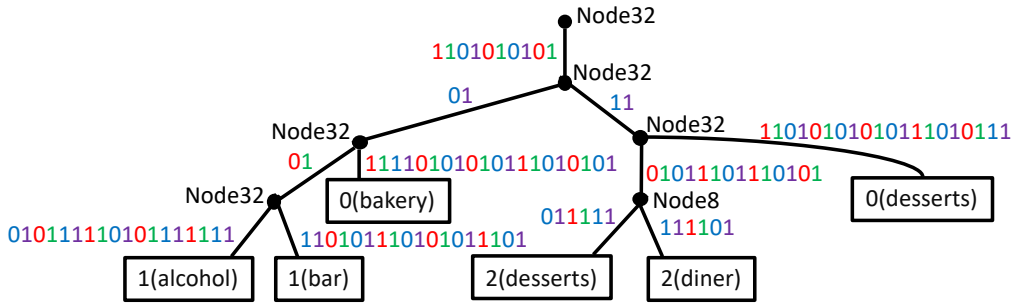

Figure 3.2: STILT index illustration with three documents as shown in Table 3.1 (maximum depth restricted to 32 bits).

3.4 Insertion

The STILT index is built by iterative insertion. The process has three steps, as described in Algorithm 1. We first calculate the path for the key (line 2) by combining the hashes according to the schedule. Then, we acquire the appropriate leaf node (line 6). Finally, we append the mapping from key to id to the leaf node's list (line 7).

Algorithm 1: Insert a key into the STILT index.

```
1 Function stilt_insert(key, id)
2     path ← combine(f_lat(key.latitude),
3                    f_lon(key.longitude),
4                    f_word(key.word),
5                    f_time(key.time))
6     node ← getsert(root, 0, path)
7     insert(node, (key → id))
```

Acquiring a node from the STILT index for insertion is described in algorithm 2. It is a continuation (line 16) function [11]; it returns lambda functions representing the next piece of work until it is finished, at which point it returns a result. The algorithm descends into the trie until a leaf node is found (line 2), or until it is at the proper location to create the desired node (lines 14 and 20).

Splitting an edge to create a new leaf node is described in algorithm 3; it involves three steps. First, we create a common node at the length where the existing path and the new path diverge (line 3). Then, we create an edge on the common node to which we attach the preexisting node such that its

Algorithm 2: Get a specific leaf node from the STILT index.

```
   Output: The node at path; it is created if it does not exist.
 1 Function getsert (node, depth, path) : LeafNode
 2     if depth = 64 then return node;
 3     go_right ← msb(path)
 4     depth ← depth + 1
 5     path ← path << 1
 6     synchronized ( node )
 7         edge ← get_edge(node, go_right)
 8         if edge does not exist then create it
 9             edge ← new Edge
10             edge.node ← new Node
11             edge.length ← 64 − depth
12             edge.path ← path
13             set_edge(node, go_right, edge)
14             return edge.node
15         else if edge matches path then continue search
16             continue with getsert(edge.node,
17                                   depth + edge.length,
18                                   path << edge.length)
19         else split the edge
20             return split(edge, depth, path)
```

path remains unchanged (line 7). Finally, we create a second edge on the common node to which we attach a new leaf node which matches the new path (line 12). We return the new leaf as it is the destination of the path.

Algorithm 3: Split an edge to create a leaf.

```
1  Function split (edge, depth, path) : LeafNode
2      (node_o, length_o, path_o) ← edge
       // find the length of the common path
3      length_c ← clz(xor(path, edge.path))
       // create and link a common node
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)
       // create and link new node
12     node_n ← new LeafNode
13     edge_n ← new Edge
14     edge_n.node ← node_n
15     edge_n.length ← 64 − depth − length_c − 1
16     edge_n.path ← path << (length_c + 1)
17     set_edge(node_c, msb(path), node_n)
18     return node_n
```

3.5 Range Search

We begin with a general definition of range search, then describe our implementation of it.

3.5.1 Definition

Definition 3.5.1. *Given a set of documents O, a spatio-temporal textual range search finds and reports the set X of documents matching a query q as follows:*

$$X \equiv \{\forall x \in X : |q.W \cap x.W| > 0, \\ d(x.\ell, q.\ell) \leq r, q.t_{lo} \leq o.t \leq q.t_{hi}\} \tag{3.1}$$

Each document $o = (id, \ell, W, t)$ in the set O of indexed documents has an id, a location ℓ, a list of words W, and a time t (e.g. the time and date the document was inserted into the database). A query $q = (\ell, r, W, t)$ has a point of interest ℓ, a radius r for the search disk centered at ℓ, a set of search terms W, a time range $t = (t_{lo}, t_{hi})$. The function $d(\ell_0, \ell_1)$ calculates the distance between two locations.

3.5.2 Implementation

Our spatio-temporal textual range search algorithm is presented in algorithm 4. The search query (ℓ, r, W, t) is applied to each keyword in the query (line 4) and is performed in two steps. First, we find every leaf node n which intersects the circular region (ℓ, r) and the time range t, and may contain at least 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 disk defined by (ℓ, r), w' is in W, and t' is in t. In our implementation, we perform the these

Algorithm 4: Range search using STILT

```
   Output: All the documents matching the search criteria.
1  Function range_search_stilt(root, query) : List
2      (area, words, time) ← query
3      ids ← ∅
4      foreach w in words do
5          ids ← ids ∪ search_node(root, (area, w, time))
6      return ids
```

steps simultaneously (line 5) to reduce memory usage, which we describe in algorithms 5 and 6.

Searching for query-matching entries is defined in Algorithms 5 and 6. They 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 23) contained in the leaf and returns them (line 25). If it is passed a non-leaf node, it determines which edges are worth searching (lines 9 and 14) and initiates `search_edge` with the modified range as appropriate. The `search_edge` function processes the edge's path (line 3) to determine range at the node. If the node is within the range (line 9), `search_node` is initiated with the modified range (line 11).

Algorithm 5: Search node and children for query matching entries.

```
1  Function search_node (node, query, depth, range) : List
2      if depth < 64 then not a leaf
3          mode ← depth mod 4
4          depth ← depth + 1
5          synchronized ( node )
6              edge_l ← node.leftEdge
7              edge_r ← node.rightEdge
8          range ← get(range, mode)
9          if query intersects lower_half(range) then
10             range′ ← clone(range)
11             set(range′, mode, lower_half(range))
12             l ← search_edge(edge_l, query, depth, range′)
13         else l ← ∅
14         if query intersects upper_half(range) then
15             range′ ← clone(range)
16             set(range′, mode, upper_half(range))
17             u ← search_edge(edge_r, query, depth, range′)
18         else u ← ∅
19         return l ∪ u
20     else is a leaf
21         ids ← ∅
22         foreach m in node.mappings do
23             if m.key matches query then
24                 ids ← ids ∪ m.id
25         return ids
```

Algorithm 6: Process edge before searching node.

```
1  Function search_edge (edge, query, depth, range) : List
2      path ← edge.path
3      foreach i in [0 .. edge.length) do adjust range
4          mode ← (depth +i) mod 4
5          r ← get(range, mode)
6          if msb(path << i) then r ← upper_half(r)
7          else r ← lower_half(r)
8          set(range, mode, r)
9      if range matches query then
10         depth ← depth + edge.length
11         search_node(edge.node, query, depth, range)
```

3.6 Lazy Compression

Besides the compression described in section 3.2, STILT can also perform compression on demand to further reduce memory usage. The process involves doing a tree walk of the trie while swapping existing non-leaf nodes for equivalent, smaller copies when possible. There are two conditions to satisfy for a node to be a compressible: it must have two children, and the largest path length of its children must be no larger than the maximum path length of a smaller node implementation. For example, if we have a `Node64` with children at lengths 3 and 15, it can be replaced with an equivalent `Node16` since this smaller implementation can represent path lengths up to 17; conversely, if a `Node64` has children at lengths 1 and 35, it cannot be compressed since the next smallest implementation (`Node32`) is unable to represent a path of length 35.

3.7 Concurrency

The STILT index supports concurrent insertions with fine-grained locking. Performing insertions and searches concurrently is supported, but it is only partially isolated. A search is guaranteed to include every document inserted before it is initiated; however, it is undefined whether a document being inserted while a search is being executed will be included. This is of no consequence in the case of streaming data; however, if necessary, a master read-write lock can ensure full isolation by preventing insertions from occurring during search operations.

Compression can be performed concurrently with searching; however, in our implementation, compression and insertion cannot be performed concurrently. This limitation is caused by our use of block-based locking, which forces an insertion thread to release a parent node's lock before acquiring the lock for one of its children. If a compression thread were to swap the child node during this lock-free period, the insertion will continue on a now-disconnected node, breaking the index. If we used non-block-based locking, the insertion thread could acquire the child's lock before releasing the parent's, preventing the compression thread from disrupting the insertion process.

Chapter 4

The STILT System

Our system is composed of an index for key searching and an LSM-tree [27, 37] to store documents, as illustrated in Figure 4.1. Every document

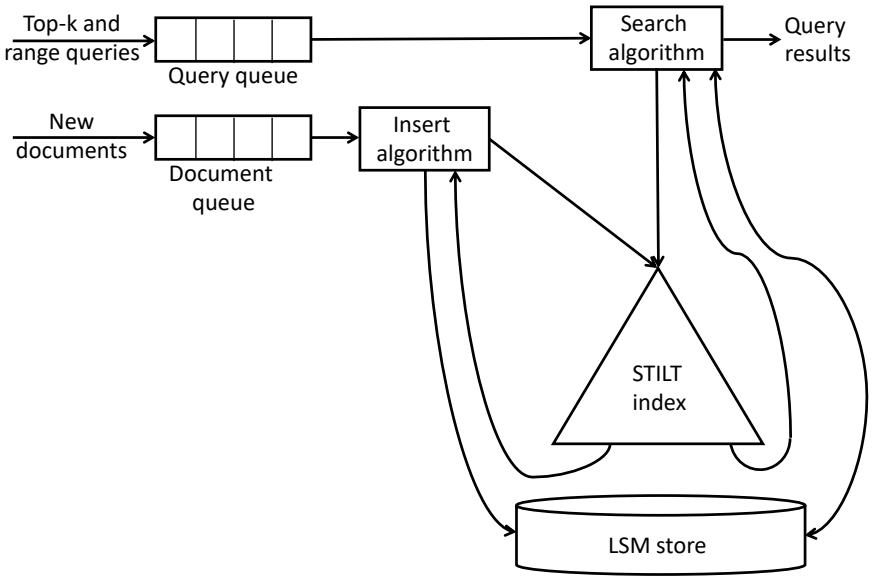

Figure 4.1: STILT architecture.

in the system has a unique id associated with it. The LSM-tree maintains ($id \rightarrow o$) mappings, and the index stores (key $\rightarrow id$) mappings. A key is composed of a spatial coordinate, a timestamp, and a single word. A document $o = (id, \ell, W, t)$ has $|W|$ keys. The following sections describe how documents are ingested and how they are queried.

4.1 Data Ingestion

The system can be built iteratively and concurrently. The insertion of a document o has three steps, as shown in Algorithm 7. First, we generate a unique integer $id \in \mathbb{Z}^+$ (line 2). Second, we insert the mapping ($id \rightarrow o$) into the LSM-tree (line 3). Third, for each of the $|W|$ 4-dimensional keys in o, we insert the mapping (key $\rightarrow id$) into the index (line 5).

Algorithm 7: Insert a document into the system

Input : The backing LSM-tree store, the index, and a document.
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)
```

If needed, the system supports bulk indexing. During bulk indexing, the id generation and LSM-tree insertion operations are performed iteratively, while (key $\rightarrow id$) mappings are gathered into a collection to be processed

after all the LSM-tree insertions are completed. This feature was necessary to accommodate ST2I.

4.2 Range Query

The range search algorithm for the system simply involves running range search on the index, which yields an id set, then fetching the matching documents from the LSM-tree [27].

4.3 Top-K Query

4.3.1 Definition

Definition 4.3.1. *A spatio-temporal textual top-k search extends the range search above to find and report the set X of the k highest ranked documents matching a query q. A ranking scheme is used to assign scores and order the documents in X.*

The top-k query $q = (\ell, r, W, t, k, c)$ adds the parameters k and c where k is the number of ranked documents to return, and c is the maximum number of increases in radius. We use a priority queue to find the k highest ranked documents, as shown in algorithm 8. A unified ranking scheme, explained in equation 4.5 in Section 4.3.2, combines spatial, temporal, and textual components into a single score $S(q, o) \in [0 \,..\, 1]$.

Algorithm 8: Top-K search

Input : A search criteria, a result-size k, the LSM-tree, the index, and the maximum number of iterations c.

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

```
1  Function topk_search(query, k, data, index, c) : List
2      radius ← query.radius
3      results ← new PriorityQueue(k)
4      foreach i in [1 .. c] do
5          clear(results)
6          query.radius ← radius * i
7          documents ← range_search(index, query)
8          foreach o in documents do
9              insert(results, o → score(o))
10         if results.size == k then
11             inner_min ← worst score in results
12             outer_max ← best score outside of radius * i
13             if inner_min > outer_max then return results
14     return results
```

4.3.2 Implementation

We describe our spatio-temporal textual top-k search algorithm in algorithm 8. It takes the same parameters as range search, but adds extra parameters $k, c \in \mathbb{Z}^+$. Top-k search takes up to c iterations to compute its result. We increase the search radius linearly (line 6) with each iteration until we either know we have the actual top k results (line 13) or until we have performed c iterations (line 14). We can determine that our result is correct before completing c iterations if two conditions are satisfied: we have already gathered at least k results (line 10), and the score of our worst document is superior to the best document score possible outside the current

search radius (line 12). This hypothetical best scoring document can be calculated by scoring a theoretical document on the periphery of the current search radius with a perfect score in non-spatial dimensions. To calculate the score $S(q, o)$ for a query q and document o (line 9), we begin by calculating three independent scores based on spatial, textual, and temporal similarity. Spatial scoring is based on distance from the query location $q.\ell$ (Equation 4.1), while temporal scoring is based on recency of the document time $o.t$ within the query time range (Equation 4.2). For textual scoring, we use term frequency-inverse document frequency (TF-IDF, equation 4.4) to score individual words, then combine the word scores using cosine similarity (equation 4.3). We combine the three scores using weights, as seen in equation 4.5. We chose even weights in out evaluation i.e. $\alpha, \beta, \gamma = \frac{1}{3}$.

$$S_s(q, o) = 1 - \frac{d(q.\ell, o.\ell)}{q.r} \tag{4.1}$$

$$S_t(q, o) = 1 - \frac{o.t - q.t_{min}}{q.t_{max} - q.t_{min}} \tag{4.2}$$

$$S_w(q, o) = \frac{\sum\limits_{w \in q.W} \text{tf-idf}(w, o)\text{tf-idf}(w, q)}{\sqrt{\sum\limits_{w \in q.W} \text{tf-idf}(w, o)^2} \sqrt{\sum\limits_{w \in q.W} \text{tf-idf}(w, q)^2}} \tag{4.3}$$

$$\text{tf-idf}(\tau, p) = \frac{|\tau \cap p.W|}{|p.W|} * \log \frac{|O|}{|\forall o \in O : \tau \in o.W|} \tag{4.4}$$

$$\begin{aligned} &\textbf{Given } \text{weights } \alpha, \beta, \gamma \in [0 \,..\, 1] : \alpha + \beta + \gamma = 1 \\ &S(q, o) = \alpha S_s(q, o) + \beta S_t(q, o) + \gamma S_w(q, o) \end{aligned} \tag{4.5}$$

Chapter 5

Evaluation

In this chapter, we evaluate our index in various settings using the system we define in chapter 4. STILT was compared against the state-of-the-art integrated index, ST2I. In fact, we extended ST2I to support parallel index construction and concurrent query execution, which are not supported by the original ST2I as described in [12]. Although our evaluation of STILT was primarily against ST2I (extended), we also considered the I^3 index [38], which is considered to be one of the fastest spatio-textual indexes. Since I^3 does not support the temporal aspect, we modified I^3 to support spatio-temporal textual top-k query.

We begin by describing our experimental setup, including a description our data sets, query sets, and the environment in which we ran our tests. We then describe our testing methodology. Finally, we reveal our experimental results relating to index construction and query processing, respectively.

Table 5.1: Datasets

Dataset name	Num. of objects	Average num. of keywords	Max num. of keywords	Average num. of words
Spaten	1,523,849	23.26	300	127.88
Tw20mi	20,000,000	5.70	70	28.89
Wikipedia	280,000	1018.39	62241	6115.17

5.1 Experimental setup

In this section we discuss the datasets, query sets, and environment setup.

5.1.1 Datasets

We used three real world datasets: Tw20mi (Twitter), Wikipedia [36], and Spaten [8]. The Twitter dataset consists of 20 million recorded tweets with 28.9 words per tweet, on average. The Wikipedia dataset, extracted from the official Wikipedia dump, has 280,000 articles with an average of 6,115 words per article. Spaten is a collection of documents with check-ins at real points of interest (such as restaurants) and associated user comments, crawled from TripAdvisor; it consists of 1.5 million documents with an average of 128 words per review. The documents in all datasets consist of geo-location, a timestamp, and a list of words. We provide a more thorough summary in Table 5.1.

Table 5.2: Parameter settings

Parameter	**Value Set**
Datasets	Tw20mi, Spaten, **Wikipedia**
Number of queries	**1000**
k	**5**, 10, 15, 20, 25
Temporal range (in weeks)	1, 2, 3, 4, 5, **all**
Query radius	1 km, **10 km**, 100 km
Number of query keywords	1, 2, 3, 4, **5**
Number of threads	**1**, 2, 4, 8, 16

5.1.2 Query sets

Our query sets consist of 1000 queries each. The query sets were created randomly in such a way that 25% of the queries in each set are guaranteed to return results. The rest of the queries were randomly generated using values existing in the dataset in such a way at least one document is guaranteed to match in each dimension. Note that it is incredibly unlikely that a document will match in all dimensions for these randomly generated queries.

5.1.3 Environment setup

The experiments were conducted on a machine with 16 AMD Opteron processing cores and 128 GB memory. Our implementations are written in Java 9 and were run on Ubuntu 16.04 using Java HotSpot 64-bit Server 18.3.

5.2 Methodology

Table 5.2 describes all variable parameters and shows the default values in **bold**. We measured construction time and memory usage with the default settings. Each query-set experiment changes one variable while keeping all other variables at their default value. Every data point was measured three times, after one warm up run. Results shown in Figures 5.5 through 5.10 include the standard deviation as well as the average.

Experiments concerning time were measured using within the application itself using `System.nanoTime`. The experiment concerning size was measured using the `Runtime` object in Java; this means we measured Java's heap size, not the total memory used by the application.

5.3 Index Construction Results

This section describes how long it takes to build each index, and how much memory is used in the process.

5.3.1 Index Construction Execution Time

Figure 5.1 shows the index building time for STILT, ST2I and I^3, for each of the datasets. In every scenario, STILT outperforms ST2I and I^3. For instance with Tw20mi, STILT takes 568.7 seconds, while ST2I takes 1,450.6 seconds and I^3 takes 2,390 seconds. Spaten is a smaller dataset, so its index

construction time is relatively smaller for all three indexing methods. With the Wikipedia dataset, STILT is 4.8× faster than ST2I and 6.5× faster than I^3. STILT's speed can be attributed to the fact that it resides entirely in memory and does not rely on any temporary data structure during the build process. On the other hand, ST2I and I^3 both rely on temporary, in-memory data structures, and they both write a considerable portion of the temporary data structures to disk, which is a slow process.

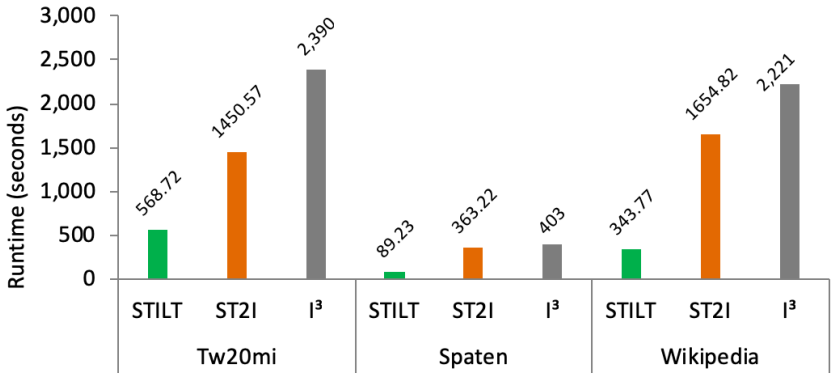

Figure 5.1: Index construction time

5.3.2 Index Memory Usage

Figure 5.2 shows the minimum memory required to build each of the indexes. ST2I utilizes memory-mapped files to store its `KBlocks`, which consumes memory not monitored by the JVM; to compensate for this, we included the memory-mapped file's size in light green. Taking into account ST2I's total memory usage, STILT used the least memory in all three cases. STILT's surprisingly low memory usage while being built is due to our adaptive node

construction and our ability to build the index in-place, without any temporary structures. We did not use STILT's lazy compression in the figure 5.2 test, which would trade-off increased processing time for reduced memory usage.

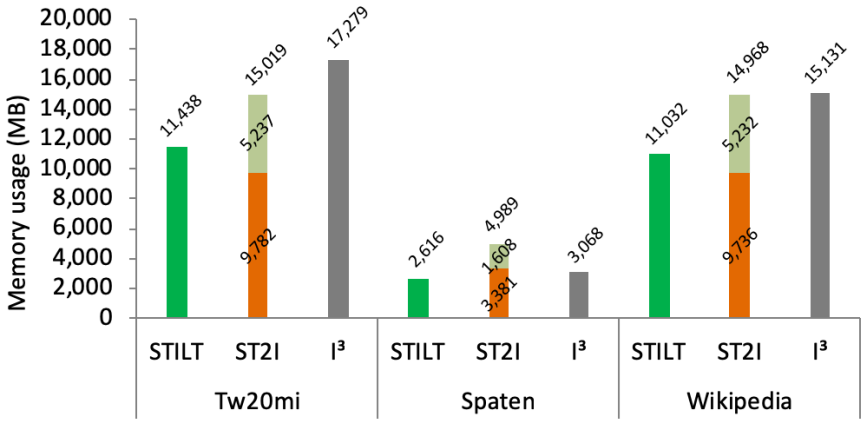

Figure 5.2: Index memory usage

5.4 Query Performance Results

We evaluated the performance of both top-k search and range search. We varied all parameters as described in section 5.1.3. Some parameters are compatible with both range search and top-k, while others (like the k parameter) only apply to one of them.

5.4.1 Top-k and Range Search: Vary Datasets

We first evaluated the performance of top-k search with STILT, ST2I and I^3, while varying the datasets, as shown in figure 5.3. STILT was the fastest in all

cases; for example, with the TW20mi dataset, STILT was about twice as fast as ST2I, and 47 times faster than I^3. Due to the poor relative performance of I^3, we omit results involving I^3 in subsequent experiments.

In figure 5.4, we evaluated the range search performance of STILT and ST2I, while varying the datasets. STILT always performs better than ST2I; for example, with the Wikipedia data set, STILT is 4.8× faster than ST2I. Note that no result is presented for I^3, as range search is not supported by I^3.

STILT's amazing performance in range search is likely due to its ability to search without accessing disk. Unlike with ST2I, which uses disk for it's KBlock structure, STILT lets the LSM-tree use the disk completely uncontested. STILT's less impressive, though still superior, performance in top-k can be attributed to the fact that documents must be retrieved from disk to be scored; this adds a considerable number of disk requests which makes the LSM-tree a performance bottleneck.

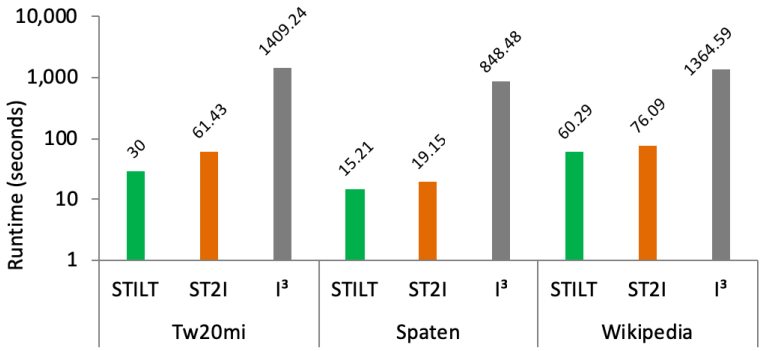

Figure 5.3: Top-k search: vary datasets

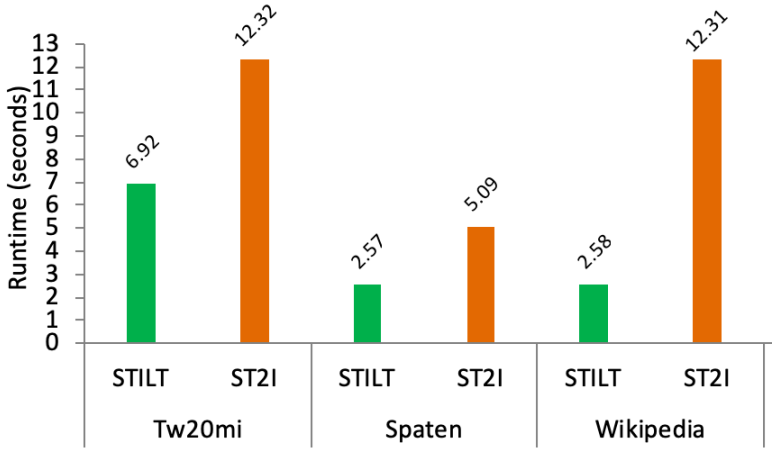

Figure 5.4: Range search: vary datasets

5.4.2 Top-k and range search: vary time range

We varied the time range from 1 to 5 weeks by starting with 1 week as the first interval, and then adding 1 week until the largest interval. STILT performs significantly better than ST2I. The top-k search results (Figure 5.5) show that for a time range of 5 weeks, STILT was 1.45× faster than ST2I. The reason why ST2I is faster for a 2 week time range is unknown, but may be due to lucky distribution among the KBlocks. As for the range search (Figure 5.6), STILT was about 4.4× faster than ST2I for a time range 5 weeks. As indicated by the relatively large standard deviation error bars, ST2I can exhibit significant variation in its results. We surmise that these large standard deviations might be due to the use of memory-mapped disk I/O which is entirely controlled by the operating system.

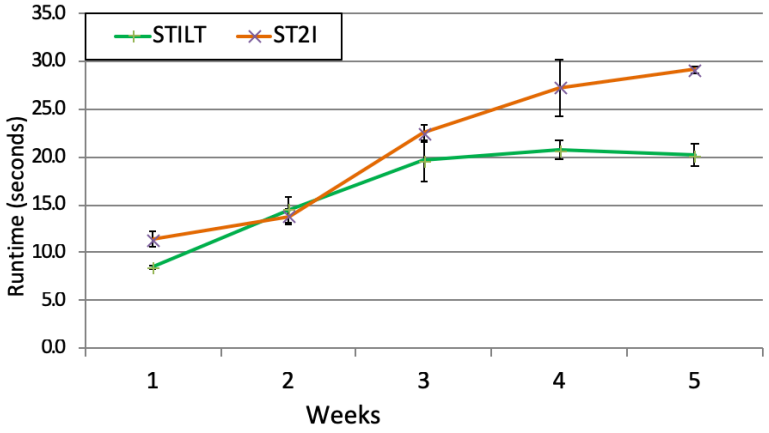

Figure 5.5: Top-k search: vary time-range

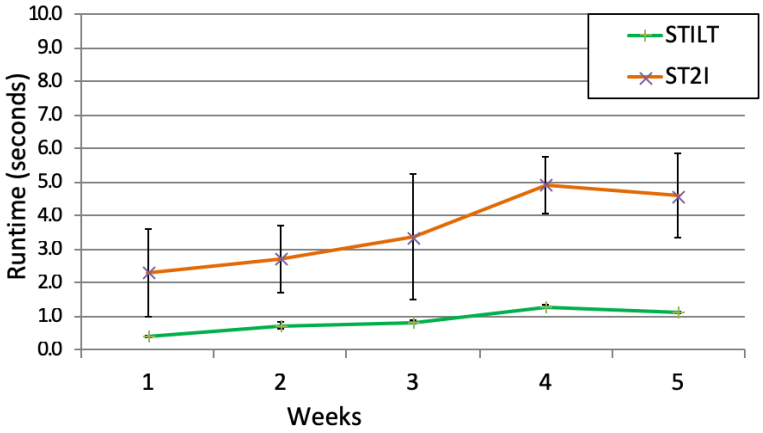

Figure 5.6: Range search: vary time range

5.4.3 Top-K Search: Vary K

We evaluate top-k search query by varying k. STILT always performs better than ST2I as indicated in figure 5.7. For instance, with k=25, STILT is about 1.2× faster. We can also see that increasing k does not significantly affect search time.

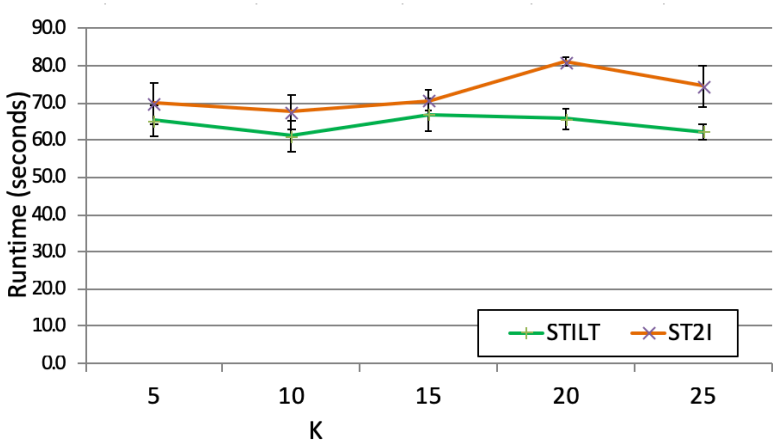

Figure 5.7: Top-k search: vary k

5.4.4 Top-k search: vary number of keywords

For the top-k search query, we vary the number of keywords from 1 to 5 and evaluate STILT and ST2I performance. As shown in figure 5.8, the number of keywords selected from our query set is not a significant factor in performance. When the number of keywords is 5, STILT is 8.0% faster than the average of its other four results; there are no obvious reasons why this would be so, especially given the small standard deviation.

5.4.5 Range search: vary radius

For range search query, we vary the search radius ranging from 1km to 100km. As shown in Figure 5.9, the search radius is a very significant factor in performance. The execution time ratios (ST2I / STILT) are 9.0, 3.2, and 2.2 for 1km, 10km, and 100km, respectively. The number of results returned increases dramatically as the radius increases; as such, we surmise that the

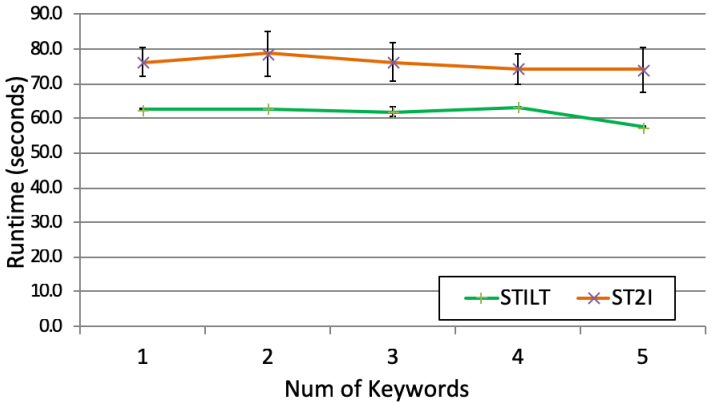

Figure 5.8: Top-k search: vary number of keywords

decrease in the performance ratio is a result of the reporting cost (fetch from LSM-tree) dominating the execution time.

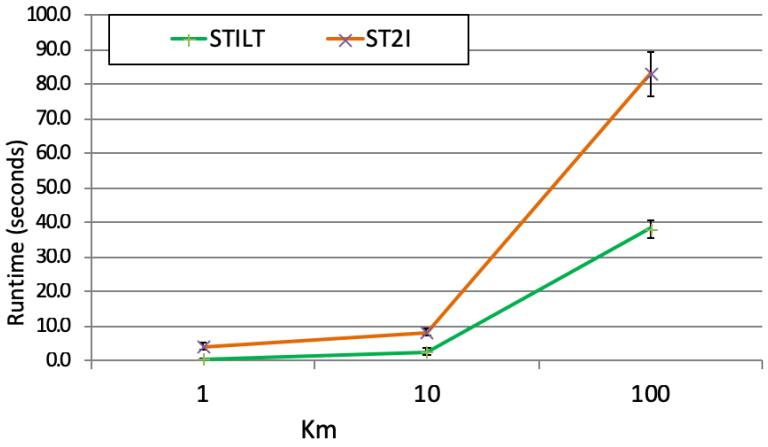

Figure 5.9: Range search: vary radius

5.4.6 Top-k search: vary number of threads

All the previous experiments involving query performance evaluation were conducted in a single-threaded setup even though STILT supports concur-

rent query execution. As mentioned in Section 5, we compare an extended ST2I implementation, in which we implemented concurrent query execution support.

Figure 5.10 shows top-k search query execution times for 1, 2, 4, 8 and 16 threads. In single thread execution, STILT was about 1.26× faster than ST2I; with 16 threads, STILT was 1.6× faster than ST2I. This shows that ST2I does have potential as a multi-threaded index, but that it does not scale any faster than STILT does. This means that ST2I will not overtake STILT in performance on multi-core machines.

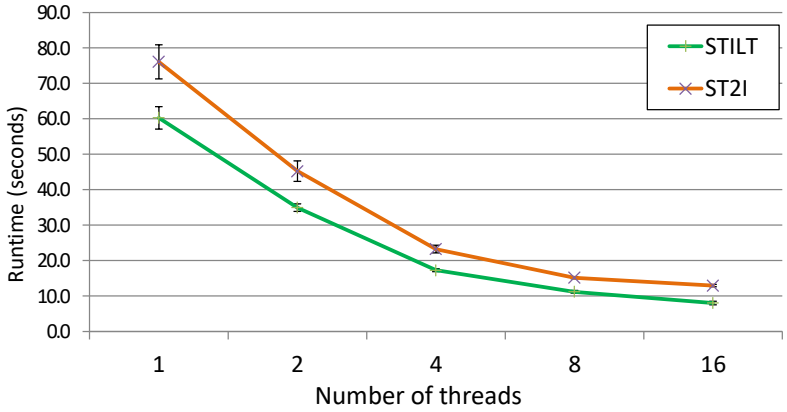

Figure 5.10: Top-k search: vary number of threads

Chapter 6

Conclusion

With the continuous growth of data that includes text, time, and location components, it is important to develop efficient indexes to support spatio-temporal textual search; however, other than ST2I [12], no index incorporates all three data components in a single structure or uses a unified ranking scheme to score the documents. On the other hand, ST2I has a static indexing approach that requires rebuilding the index when new data needs to be ingested. It also does not support parallel operations and supports a limited character set.

6.1 Contributions

To address these limitations, we introduced a parallel in-memory integrated index called STILT, which supports efficient spatio-temporal textual range

search and top-k search using a combined ranking scheme. STILT is a multi-dimensional binary trie index that is suitable for modern multi-core machines and uses an adaptive node structure to achieve memory efficiency. It also supports simultaneous insertion and searching for efficient processing of continuous data streams. We also provide an easily extensible character mapping function which can handle any desired character set, with a clear and useful policy for unrecognized characters. With extensive experimental evaluation, we demonstrated that STILT significantly outperforms the state-of-the-art spatio-temporal textual integrated index.

6.2 Future Work

As future work, we would like to explore a few ideas. First, we would like to store enough information in main memory to compute the top-k score without retrieving candidate documents. To do this, we would need to know the frequency of found keywords as well as the total number of words in the documents. Second, we want to reduce redundant information in keys. Each of a given document's keys shares the same latitude, longitude, and temporal component; the only unique component is the word. We can extract the duplicate components into a shared object, which would result in significant savings for datasets like Wikipedia which have over 1,000 keys per document. Third, we would like to experiment with different mapping functions and path schedules. These two properties of the STILT index define

its distribution and have a very strong impact on its performance; e.g. we could use a Unicode-based [33] character mapping scheme to support virtually every document ever created. Finally, we plan to develop techniques for real-time compression of node type during insertion, rather than relying on lazy compression.

References

[1] Daniel C. Andrade, João B. Rocha-Junior, and Daniel G. Costa, *Efficient processing of spatio-temporal-textual queries*, WebMedia, 2017, pp. 165–172.

[2] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis, *Hot: A height optimized trie index for main-memory database systems*, International Conference on Management of Data (SIGMOD), 2018, pp. 521–534.

[3] Lisi Chen, Gao Cong, Xin Cao, and K.-L Tan, *Temporal spatial-keyword top-k publish/subscribe*, International Conference on Data Engineering (ICDE), 2015, pp. 255–266.

[4] Lisi Chen, Gao Cong, Christian S. Jensen, and Dingming Wu, *Spatial keyword query processing: an experimental evaluation*, Proceedings of the VLDB Endowment (PVLDB), 2013.

[5] Su Chen, Beng Chin Ooi, Kian-Lee Tan, and Mario A. Nascimento, *St2b-tree: A self-tunable spatio-temporal b+-tree index for moving objects*, International Conference on Management of Data (SIGMOD), 2008, pp. 29–42.

[6] Maria Christoforaki, Jinru He, C. Dimopoulos, Alexander Markowetz, and Torsten Suel, *Text vs. Space: Efficient Geo-search Query Processing*, Conference on Information and Knowledge Management (CIKM), 2011.

[7] Gao Cong, Christian S. Jensen, and Dingming Wu, *Efficient retrieval of the top-k most relevant spatial web objects*, Proceedings of the VLDB Endowment (PVLDB) (2009), 337–348.

[8] Thaleia Dimitra Doudali, Ioannis Konstantinou, and Nectarios Koziris, *Spaten: A spatio-temporal and textual big data generator*, International Conference on Big Data (IEEE Big Data), 2017, pp. 3416–3421.

[9] *GSM Association real-time tracker*, https://www.gsmaintelligence.com/.

[10] Jinru He and Torsten Suel, *Faster temporal range queries over versioned text*, International Conference on Research and Development in Information (SIGIR), 2011, pp. 565–574.

[11] R. Hieb and R. Kent Dybvig, *Continuations and concurrency*, Symposium on Principles and Practice of Parallel Programming (PPoPP), 1990, pp. 128–136.

[12] Tuan-Anh Hoang-Vu, Huy T. Vo, and Juliana Freire, *A unified index for spatio-temporal keyword queries*, Conference on Information and Knowledge Management (CIKM), 2016, pp. 135–144.

[13] *Internet Live Stats*, http://internetlivestats.com/, 2019, Accessed March 20, 2019.

[14] Christian S. Jensen, Dan Lin, and Beng Chin Ooi, *Query and update efficient B+-tree based indexing of moving objects*, Proceedings of the VLDB Endowment (PVLDB), 2004, pp. 768–779.

[15] Ali Khodaei, Cyrus Shahabi, and Amir Khodaei, *Temporal-textual retrieval: Time and keyword search in web documents*, International Journal of Next-Generation Computing (IJNGC) (2012).

[16] Ali Khodaei, Cyrus Shahabi, and Chen Li, *Hybrid indexing and seamless ranking of spatial and textual features of web documents*, International Conference on Database and Expert Systems Applications (DEXA), 2010, pp. 450–466.

[17] Kalev Leetaru, *Visualizing seven years of twitter's evolution: 2012-2018*, Forbes (2019), Accessed March 29, 2019.

[18] Viktor Leis, Alfons Kemper, and Thomas Neumann, *The adaptive radix tree: Artful indexing for main-memory databases*, International Conference on Data Engineering (ICDE), 2013, pp. 38–49.

[19] Z. Li, K. C. K. Lee, B. Zheng, W. Lee, D. Lee, and X. Wang, *Ir-tree: An efficient index for geographic document search*, IEEE Transactions on Knowledge and Data Engineering (TKDE) (2011), 585–599.

[20] Dan Lin, Christian S. Jensen, Beng Chin Ooi, and Simonas Šaltenis, *Efficient indexing of the historical, present, and future positions of moving objects*, Mobile Data Management (MDM), 2005, pp. 59–66.

[21] Amr Magdy, Louai Alarabi, Saif Al-Harthi, Mashaal Musleh, Thanaa M. Ghanem, Sohaib Ghani, and Mohamed F. Mokbel, *Taghreed: A system for querying, analyzing, and visualizing geotagged microblogs*, International Conference on Advances in Geographic Information Systems (SIGSPATIAL), 2014, pp. 163–172.

[22] Amr Magdy, Mohamed F. Mokbel, Sameh Elnikety, Suman Nath, and Yuxiong He, *Mercury: A memory-constrained spatio-temporal real-time search on microblogs*, International Conference on Data Engineering (ICDE), 2014, pp. 172–183.

[23] Ahmed R. Mahmood, Ahmed M. Aly, Thamir Qadah, El Kindi Rezig, Anas Daghistani, Amgad Madkour, Ahmed S. Abdelhamid, Mohamed S. Hassan, Walid G. Aref, and Saleh Basalamah, *Tornado: A Distributed Spatio-textual Stream Processing System*, Proceedings of the VLDB Endowment (VLBD) (2015), 2020–2023.

[24] Donald R. Morrison, *PATRICIA–Practical Algorithm To Retrieve Information Coded in Alphanumeric*, Journal of the ACM (JACM) (1968), 514–534.

[25] L. Morrison and J. Morgan, *Who tweets with their location? understanding the relationship between demographic characteristics and the use of geoservices and geotagging on twitter*, Public Library of Science One (PLOS One) (2015).

[26] Bradford G. Nickerson and Qingxiu Shi, *On k-d range search with patricia tries*, SIAM Journal on Computing (2008), 1373–1386.

[27] O'Neil, Patrick and Cheng, Edward and Gawlick, Dieter and O'Neil, Elizabeth, *The Log-structured Merge-tree (LSM-tree)*, Acta Informatica (1996).

[28] Dieter Pfoser, Christian S. Jensen, and Yannis Theodoridis, *Novel Approaches in Query Processing for Moving Object Trajectories*, Proceedings of the VLDB Endowment (PVLDB), 2000.

[29] Suprio Ray and Bradford G. Nickerson, *Improving parallel performance of temporally relevant top-k spatial keyword search*, SIGSPATIAL Workshop on Recommendations for Location-based Services and Social Networks (LocalRec), 2018, pp. 5:1–5:4.

[30] João B. Rocha-Junior, Orestis Gkorgkas, Simon Jonassen, and Kjetil Nørvåg, *Efficient processing of top-k spatial keyword queries*, International Symposium on Spatial and Temporal Databases (SSTD), 2011, pp. 205–222.

[31] Yufei Tao and Dimitris Papadias, *Mv3r-tree: A spatio-temporal access method for timestamp and interval queries*, Proceedings of the VLDB Endowment (PVLDB), 2001, pp. 431–440.

[32] Yufei Tao, Dimitris Papadias, and Jimeng Sun, *The TPR*-tree: An Optimized Spatio-temporal Access Method for Predictive Queries*, Proceedings of the VLDB Endowment (PVLDB), 2003, pp. 790–801.

[33] *Unicode*, https://unicode.org, 2019, Accessed May 5, 2019.

[34] Subodh Vaid, Christopher B. Jones, Hideo Joho, and Mark Sanderson, *Spatio-textual indexing for geographical search on the web*, International Symposium on Spatial and Temporal Databases (SSTD), 2005.

[35] Xiang Wang, Ying Zhang, Wenjie Zhang, Xuemin Lin, and Wei Wang, *Ap-tree: efficiently support location-aware publish/subscribe*, Proceedings of the VLDB Endowment (PVLDB) (2015), 823–848.

[36] *Wikipedia Dump*, https://dumps.wikimedia.org/enwiki/latest/, 2018, Accessed October, 2018.

[37] Jun Yuan, Yang Zhan, William Jannen, Prashant Pandey, Amogh Akshintala, Kanchan Chandnani, Pooja Deo, Zardosht Kasheff, Leif Walsh, Michael A. Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter, *Optimizing every operation in a write-optimized file system*, USENIX Annual Technical Conference (ATC), 2016.

[38] Dongxiang Zhang, Kian-Lee Tan, and Anthony K. H. Tung, *Scalable top-k spatial keyword search*, International Conference on Extending Database Technology (EDBT), 2013, pp. 359–370.

Appendix A

Source Code Statistics

A.1 Overall Summary

Overall Code Summary

Files	Lines	Description
25	5,618	STILT Index
35	4,241	STILT System
17	2,172	ST2I Index
2	665	Linear Index
79	12,696	**Total**

A.2 Breakdown By Category

STILT Index Source Code Summary

Files	Lines	Description
4	3,483	STILT index
5	440	Continuation
3	279	Experiment launcher
3	265	Visualizer
10	1,151	Miscellaneous
25	5,618	**Total**

STILT System Source Code Summary

Files	Lines	Description
6	922	LSM-tree interface
6	863	Top-k search
7	784	Document processing
4	545	Range search
3	489	STILT system
9	638	Miscellaneous
35	4,241	**Total**

ST2I Source Code Summary

Files	Lines	Description
9	1,477	ST2I index
5	630	KBlock managers
3	65	Experiment launcher
17	2,172	**Total**

Baseline Source Code Summary

Files	Lines	Description
1	350	Linear Index
1	315	Experiment
2	665	**Total**

Vita

Candidate's full name: Yoann S. M. Arseneau

University attended: BCS from UNB Fredericton (Sept 2007 - Apr 2016)

Submitted Publication:
Yoann Arseneau, Saransh Gautam, Bradford G. Nickerson and Suprio Ray. STILT: A Parallel Integrated Index for Spatio-temporal Textual Search. (submitted)

Conference Presentation:
Yoann Arseneau, Suprio Ray and Bradford G. Nickerson. Memory Efficient Spatio-Textual Search with Concurrent Updates. Poster at the 15th Annual UNB Faculty of Computer Science Research Exposition, Fredericton, N.B., April 11, 2018