JOURNAL OF SPATIAL INFORMATION SCIENCE
Number 24 (2022), pp. 115–156 doi:10.5311/JOSIS.2022.24.199

RESEARCH ARTICLE

Temporally relevant parallel top-k spatial keyword search

Suprio Ray and Bradford G. Nickerson

Faculty of Computer Science, University of New Brunswick, Fredericton, Canada

Received: November 12, 2021; returned: March 29, 2022; revised: May 12, 2022; accepted: June 8, 2022.

Abstract: New spatio-textual indexing methods are needed to support efficient search and update of the massive amounts of spatially referenced text being generated. Location based services using geo-tagged documents provide valuable ranked recommendations about nearby restaurants, services, sales, emergency events, and visitor attractions. Consequently, top-k spatial keyword search queries (TkSKQ) have received a lot of attention from the research community. Several spatio-textual indexes have been proposed to efficiently support TkSKQ. Some of these indexes support updates based on live document streams, but the ranking schemes employed by them do not simultaneously incorporate temporal relevance, textual similarity and spatial proximity. Moreover, existing approaches have limited or no capability to exploit parallelism with document ingestion and query execution. We present a parallel spatio-textual index, Pastri, to address the aforementioned issues. Pastri can be updated incrementally over real-time spatio-textual document streams. To support temporally relevant ranking of continuously generated document streams, we propose a dynamic ranking scheme. Our approach retrieves the top-k documents that are most temporally relevant at the time of a query execution. We implemented Pastri and we integrate it within a system with a persistent document store and several thread pools to exploit parallelism at various levels. Experimental evaluation involving real-world datasets and synthetic datasets (that we created) demonstrates that our system is able to sustain high document update throughput. Furthermore, Pastri's TkSKQ search performance is one to two orders of magnitude faster than other spatio-textual indexes.

Keywords: spatial keyword search, spatio-textual, I/O-efficient indexing, top-k, temporal relevance

1 Introduction

The confluence of the web, mobile devices, skyrocketing social media use and Location-Based Services (LBS) is contributing significantly to the Big Data era. Twitter, for example, is estimated to be receiving over 9,300 tweets per second in 2021 [2], or more than 800 million tweets per day. When tweets or other forms of social media posts are georeferenced, valuable knowledge can be obtained to support improved route planning, faster delivery of emergency services, and more effective city visitor engagement. In addition, the massive amount of data being continuously generated needs highly efficient systems that can ingest new data at a high rate, while simultaneously answering many search requests per second.

The processing of textual documents requires a way to score and rank the documents that are relevant to a particular *search criteria*, which often specifies a time component. We call this type of information retrieval *temporally relevant*, as the time a social media post or news article was posted influences how well it matches a search query that is time-sensitive. In the information retrieval domain, TF-IDF (term frequency, inverse document frequency) [41] is a measure widely used in ranking algorithms. TF-IDF evaluates the importance of a word in a collection of documents, and requires a priori knowledge of the entire dataset. The *entire corpus of documents must be available upfront* in order for the ranking algorithm to work. Therefore, TF-IDF is not suitable for real-time ranking of continuously updated geo-tagged documents. Moreover, TF-IDF does not consider the age of a document while ranking.

Novel approaches are required for efficient storage, indexing, query processing and retrieval of these massive amounts of continuously generated geotagged documents. Mahmood and Aref [28] classify spatio-textual queries as (i) filter, (ii) top-k, (iii) collective and (iv) other. Top-k spatial keyword queries (TkSKQ) retrieve up to k documents based on their textual similarity to the query keywords *and* their distance from the query location. A temporally relevant TkSKQ also considers the age or recency of a document. An example of such a query might be "presidential inauguration within 10 km of Washington DC". This query should retrieve a different set of documents in 2021 than a query about the presidential inauguration in 2017 or 2013. The importance of geo-tagged document search has given rise to a number of spatio-textual indexing approaches. Chen et al. [14] conducted a comprehensive study of 12 of them. They point out that only a few support TkSKQ, such as [18, 40, 54]. These indexes were not, however, designed for real-time document streams and as such, they all share a few limitations in this context. **First**, they need a data loading phase to ingest the data and build the index. The entire dataset must be made available a priori in order to calculate the spatial and textual relevance score needed to build the index. **Second**, these indexes do not consider the temporal relevance. These indexes use ranking functions, such as TF-IDF [41] and the language model [36], that combine a spatial and a textual relevance score. For instance, the I^3 index [54] uses the TF-IDF measure. **Third**, these spatio-textual indexes are not capable of sustaining near real-time data ingestion. Indexing systems such as I^3 and IR-tree [18] are based on tree data structures, offering limited concurrent processing that is not scalable with massive update-heavy workloads.

To address some of the above-mentioned issues, several approaches have been proposed, with a primary focus on textual data streams. Earlybird [12] was designed to enable near real-time text search, but does not support spatio-textual queries. Taghreed [26] is capable of executing arbitrary queries on microblog streams by utilizing an in-memory index supporting continuous digestion of real-time microblogs. It can execute several categories of queries such as spatio-temporal boolean range keyword search, and top-k *frequent* keyword query. However, their approach [26] does not support ranked TkSKQ or a ranking scheme that incorporates both spatial and textual relevance. Mercury [27], is an in-memory index that was designed to support real-time search using top-k spatio-temporal queries over microblog streams. Mercury's ranking scheme, however, only takes into account spatial and temporal relevance, not textual relevance. Our system shares several goals with these systems, namely, high throughput ingestion of spatio-textual document streams and ad hoc (snapshot) queries with low latency search time. Unlike these systems, however, our approach supports ad hoc TkSKQ queries and a unified ranking scheme that incorporates temporal, spatial and textual relevance.

Researchers have proposed several location-aware publish-subscribe systems to enable continuous spatio-temporal query execution over data streams. Note that a continuous query is different from an ad hoc (snapshot) query. First, a continuous query needs to be registered. Then, the query is evaluated continuously over the incoming data stream. A few indexing systems that support continuous queries over spatio-textual data streams include TaSK [13], AP-Tree [51] and FAST [29]. Tornado [30, 31] is a distributed in-memory stream processing system to execute continuous spatio-textual queries. These publish-subscribe approaches are not designed to answer ad hoc (snapshot) TkSKQ queries, as Pastri does.

Although TkSKQ queries are the main focus of this paper, our system also supports spatio-temporal textual top-k query (TkSTTQ). TkSTTQ uses a ranking scheme that incorporates spatial, temporal and textual components of a document in its overall scoring mechanism. Pastri supports temporally relevant TkSKQ and provides direct support for TkSTTQ ad hoc queries that include a temporal range restricting the search to find and rank documents within a specific time window.

To summarize the goals of our system, we support: (1) high throughput ingestion of spatio-textual data streams, (2) ad hoc (snapshot) TkSKQ and TkSTTQ queries with interactive response time, and (3) a dynamic ranking scheme that considers spatial and textual relevance along with temporal recency. To that end, we propose a dynamically ranked TF-IDF (DRTF-IDF). This is a time-based ranking scheme that adapts as continuous streams of new documents are ingested by the system [37]. In order to support updates based on real-time document streams and efficient TkSKQ over such data, we present our spatio-textual index PASTRI (PArallel Spatio-Textual adaptive Ranking-based Index). We use the non-capitalized form of PASTRI, "Pastri" in the rest of the paper.

We developed an end-to-end system that was architected to be scalable, and capable of supporting real-time document updates and concurrent query execution. The Pastri index is conceptualized as a hybrid between a tree-based index and a grid index. The spatial domain is organized as an in-memory grid and its cells are indexed by an in-memory STR (Sort-Tile-Recursive) R-tree based component. The updates are managed by a component that combines the in-memory grid with inverted lists, along with an in-memory table that is backed by an in-memory cache and on-disk persistent storage. The persistent data store is based on an extension to the log-structured merge-tree (LSM-tree) [35]. Our extension, called pLSM (partitioned LSM-tree) store, can support high data update rate that is suitable for ingesting document streams (see details in Section 5.3). Queries are processed by an in-memory tree component that performs efficient circular range search from the query location. In our approach, the textual relevance score of the documents are *not* calculated during the index building time. Pastri uses our dynamic ranking measure DRTF-IDF to calculate textual relevance score based on the keywords in a given query in real-time. This on-the-fly computation of relevance score ensures that the query results include the most recent relevant objects. We present a detailed complexity analysis of our insert and query processing algorithms and prove (see Section 6) that the amortized cost to insert one data item into our system is $\Theta(\frac{log_2 \frac{N}{BP}}{BP})$ I/Os, where N is the number of data items in the pLSM store, P is the number of LSM-trees (i.e. the number of partitions) comprising the LSM store partitions in the pLSM store, and a block of data transferred to external memory disk holds B data items.

We have implemented a prototype of our system and evaluated it with several datasets. This paper extends our previous work [37, 38], however, we have incorporated significant enhancements, including, new algorithmic details and analytical proofs, and extensive evaluation and analysis. We also present new synthetic spatio-textual datasets that we have created. Furthermore, we present experimental results with a new real-world Wikipedia dataset. Our end-to-end system delivers high performance with a single enterprise server class machine having large main memory and multiple cores. Such machines can have hundreds of GBs or a few TBs of main memory. Since our approach uses a parallel in-memory index, it could be extended to multiple nodes by exploiting high-speed networking. Our in-memory index component uses dictionary encoding to significantly reduce memory footprint. Furthermore, since our dynamic ranking scheme (DRTF-IDF) uses an exponential decay parameter (to support recency), relatively older documents can be flushed from the index and a disk-based index can be used to search over older documents. We present a thorough evaluation of our system Pastri in Section 7. We show that Pastri can ingest about 200,000 documents/second with a 20 million record dataset. In contrast, Taghreed [26] reportedly can ingest up to 32,000 microblogs/second; [26] does not report any query performance. Mercury [27] can ingest 64,000 microblogs/second and achieves a query latency around 4 msec. Recall that Mercury's ranking scheme does not incorporate textual relevance. On the other hand, Pastri uses a ranking scheme that takes both textual and spatial relevance into account. With Pastri index, our system supports efficient query processing with latencies that are comparable to those of Mercury or lower. Moreover, compared to two popular indexes, IR-tree and I^3, Pastri's update throughput and TkSKQ query performance is significantly better (see the results presented in Section 7). In Section 7.6 we show the query performance of TkSTTQ is significantly better with Pastri compared to IR-tree and I^3.

The key contributions of this paper are:

- We present a novel spatio-textual index, Pastri, that supports real-time ingestion of document streams and concurrent execution of multiple arbitrary TkSKQ and TkSTTQ queries.
- We introduce a temporally relevant dynamic ranking scheme for document streams called DRTF-IDF.
- We explore a few optimization techniques in the context of a scalable system architecture, including pLSM for high throughput data insertion and three load-balancing algorithms to handle data skew.

The rest of the paper is organized as follows. In section 2 we describe the related works. Section 3 outlines the problem statements. We present our dynamic ranking scheme in section 3.1. In section 5 our index Pastri is presented along with the algorithms for update and query processing. Then in section 6 we provide analysis of these algorithms. Section 7 presents evaluation study and finally, section 8 concludes the paper.

2 Related Work and background

In this section, we discuss related work. First, we mention previous research pertaining to spatio-textual indexing and then outline previous work on indexing spatio-textual data streams. Finally, we present work related to ranking streaming documents.

2.1 Spatio-textual Index

Spatial keyword search has been growing in importance in recent years. Consequently, researchers have proposed several spatio-textual indexes. Based on the structure of spatio-textual indexes, they can be generally classified into three categories: tree based indexes, grid-based indexes and space filling curve based indexes. Chen et al. [14] evaluated 12 of these indexes. Their evaluation study suggests that not all of these indexes support TkSKQ search. Almaslukh et al. [4] performed an experimental study of ten main-memory indexes. They were constructed as combinations of four indexing building blocks: R-tree, quadtree, grid, and inverted index. The ranking scheme used in their evaluation is based on spatial and temporal components, but it lacks a textual component.

Among the general categories of spatio-textual indexes, the IR-tree [18] is one of earliest tree based approach to be proposed. It is an R-tree based index and it supports TkSKQ. IR-tree combines an R-tree with an inverted file. Specifically, it enhances the nodes of the R-tree with summary information regarding the textual content of the node's corresponding subtree. The spatial relevance is calculated from the MBR, whereas, an upper bound of the textual relevance score is computed from the summary information. This information is utilized for the purposes of pruning search paths. Several variants of the IR-tree are also proposed by the authors, that include the DIR-tree, the CIR-tree and the CDIR-tree. The S2I [40] is another tree based index. It exploits two different approaches: one for frequent terms and the other for infrequent terms. In the case of infrequent keywords, the elements in an inverted file are stored sequentially to support efficient I/O. For frequent keywords, an aggregated R-tree (aR-tree) is utilized for better pruning. The I^3 index [54] is another spatio-textual index, which is based on a textual partitioning approach similar to that of S2I. However, for spatial indexing it utilizes a quadtree, rather than an R-tree. Choudhury et al. [16] presented an approach designed for batch processing of queries. It can take advantage of the spatial proximity of a batch of queries when searching their IR-tree based index. Hoang-Vu et al. proposed a k-d tree based index, called ST2I [24], which supports queries that combine keywords with spatial and temporal constraints. ST2I, however, does not support streaming updates. Also, ST2I does not consider a dynamic ranking scheme.

SFC-QUAD [17] is one of the few non-R-tree based indexing approaches that support TkSKQ. It is based on an inverted file in which the document id and object frequencies are compressed using the OPT-PFD algorithm [53]. The documents maintained by each inverted list are ordered by the position of the document ids in a Z-curve. SFC-QUAD also utilizes a quadtree for efficient traversal.

The grid-based indexes integrate a grid index with a text index. Among the grid-based indexes ST & TS [48] and SKIF [25] are perhaps the most well-known. ST and TS can be treated as a loose organization of spatial-first and textual-first indices, respectively. SKIF uses an inverted-file structure to store the textual information. The grid-based approaches can only support boolean range queries, but not TkSKQ, as reported by [14].

STILT [7] is a recently proposed trie-based index that interleaves text, location, and time information within a single structure. It supports spatio-temporal textual queries with any combination of dimensions involving location, time and text, including top-k spatial keyword search queries. However, it uses the TF-IDF scheme for textual ranking, which is not temporally relevant.

The goal our index, Pastri, shares with the above-mentioned indexes is to be able to support ad hoc (snapshot) top-k spatial keyword search. Pastri can be considered as a hybrid between a tree and a grid-based approach and it can support top-k spatial keyword query. Unlike Pastri, however, none of these indexes considered temporal relevance based on dynamic ranking. We also support high document update throughput with a partitioned LSM-tree based persistent data store. Moreover, our highly multi-threaded system can exploit parallelism at various levels.

Researchers investigated geo-social temporal queries [5,45] in which the social context, such as popularity among friends, is considered. Due to their focus on social aspects, they are somewhat orthogonal to our approach.

2.2 Indexing spatio-textual data streams

Motivated by the rapid rise in streaming data sources, social media and the popularity of online search, a number of research projects investigated the topic of keyword search on data streams. Markowetz et al. [32] proposed an approach for keyword search on relational data streams, which requires no knowledge about the schema. However, their approach does not consider a ranking mechanism. Also, it does not retrieve the top-k results. Cheng et al. [15] investigated how to extract representative posts from data streams, particularly with micro-blogging data.

In recent years, supporting low latency search and updates has been a major focus of the research on real-time search over massive streams. The Earlybird [12] search engine answers queries with an average latency of 50 ms that include tweets submitted 10 seconds before the search query was issued. However, Earlybird is designed for real-time textual search and it does not support spatial criteria. StreamCube [21] uses a similarity measure based on both hashtags and words in tweets from Twitter stream to automatically generate top-k event rankings. Unlike Pastri, however, StreamCube is not capable of performing user-defined top-k spatial keyword search.

Magdy et al. [26] proposed Taghreed, a microblog data management system. It supports several types of queries, for instance, finding the top-k most frequent keywords used within given spatial and temporal ranges. However, Taghreed does not support ranked top-k spatial keyword queries as defined in Section 1. Mercury [27] is another system designed for real-time search on microblogs. Although, it supports top-k spatio-temporal queries, however, its ranking scheme is based on spatial and temporal relevance. Hence, its ranking scheme does not consider textual relevance.

A key objective of Pastri is to support ad hoc (snapshot) ranked top-k spatial keyword queries. On the other hand, a continuous query is required to be registered with the system, before it can be evaluated on the incoming data stream. Therefore, publish-subscribe approaches [13, 23, 29–31, 51], which are designed for continuous spatio-temporal queries over data streams are not pertinent to this discussion.

2.3 Ranking streaming textual documents

In the information retrieval research, TF-IDF (term frequency, inverse document frequency) [41] is a very popular similarity measure used for ranking the relevance of documents. A few projects explored TF-IDF for ranking streaming documents. In [11], a modified measure called TF-PDF (term frequency, proportional document frequency) was proposed that attempts to assign significant weight to the terms related to hot topics available from newswire sources on the Web. In the context of a real-time unsupervised document clustering problem, a scheme called TF-ICF (where C is for corpus) was proposed [39]. Other ranking mechanisms have been proposed as well. For instance, for semantic-aware querying and event detection, similarity measures [8, 46] were introduced that incorporate semantic aspects, besides spatial or temporal features. However, these approaches are orthogonal to our approach.

The work that most resembles our document ranking method is the Approximate TF-IDF (ATF-IDF) proposed by Erra et al. [20]. Unlike the precise version of TF-IDF, ATF-IDF does not require a complete knowledge of the number of documents and hence it can be applied to streamed data processing. Our scheme, dynamically ranked TF-IDF (DRTF-IDF), also calculates approximate ranking. However, ATF-IDF does not consider temporal relevance.

AFIA [44] supports top-k most frequent term searches over massive geo-tagged Twitter streams by using *approximate* ranking. While this is a different problem than Pastri's TkSKQ, Pastri also uses an approximate TF-IDF ranking for text. Pastri's ranking scheme, DRTF-IDF, is an extension of our initial work [37]. In this paper we present a formalized notion of DRTF-IDF and illustration of this scheme.

3 Problem statement: temporally relevant top-k spatial keyword query

Definition 1. *Given a continuously updated set of spatio-temporal documents O, a temporally relevant top-k spatial keyword query (TkSKQ) q finds the set X of the k highest ranked matching documents, which are ranked based on equation 7 as follows:*

$$\begin{aligned} X \equiv \operatorname{sort}(\{\forall x \in X : |q.W \cap x.W| > 0, \\ d(x.l, q.l) \leq cr, |X| \leq k\}). \end{aligned} \tag{1}$$

Here, each document $o = (id, l, W, t_o)$ in the set O of indexed documents has an identifier $o.id$, a two-dimensional spatial location $o.l$, a set of terms or words $o.W$ comprising the document contents, and a time t_o when the document was defined. Further, $q = (l, W, t)$, where $q.l$ is the query spatial location, $q.W$ is the set of terms or words being searched for, and $q.t$ is the query time. An initial radius r of a search disk centered at $q.l$ is defined, as well as a constant $c \in \mathbb{N}_1$ defining the maximum number of increases in radius (if the initial search radius yields less than k matches). The distance $d(x.l, q.l)$ between the document and query locations is used to compute the spatial match score as defined in equation 2. X is a sorted set of size k (or less) consisting of the (up to) k highest ranking documents scored according to equation 7. The sorted result set X is maintained in a priority queue as described in section 5.5.

3.1 Temporally relevant dynamic ranking

Our ranking process considers the spatial location, text and time similarity of the query q to each document in O. The rank 1 document corresponds to the closest match to the query.

3.1.1 Spatial Match

The spatial match score $S(o, q) \in [0, 1]$ is defined as the distance from the query location $q.l$ to the document location $o.l$, normalized by r_i, where r_i is the radius of the current search disk, as follows:

$$S(o, q) = \begin{cases} 1 & \text{if } o.l = q.l, \\ 1 - 2((d - r_i)/r_i)^2 & \text{if } 0 < d \leq r_i/2, \\ 2(d/r_i)^2 & \text{if } r_i/2 \leq d < r_i, \\ 0 & \text{if } d \geq r_i, \end{cases} \tag{2}$$

where d is the spatial distance from $q.l$ to $o.l$. Equation 2 corresponds to a piecewise parabolic curve, which is sometimes used to define fuzzy set membership functions that approximate the sigmoid function shape.

3.1.2 Textual Match

The textual match score $T(o, q)$ accounts for the frequency of terms in documents in O, and in the query q. It is computed as follows:

$$T(o, q) = \frac{\sum_{w \in q.W} \mathit{tfidf}(w, o)\mathit{tfidf}(w, q)}{\sqrt{\sum_{w \in q.W} \mathit{tfidf}(w, o)^2}\sqrt{\sum_{w \in q.W} \mathit{tfidf}(w, q)^2}}, \tag{3}$$

where w is a specific term (e.g. a word) in the document o or the query q. Equation 3 was introduced by Salton et al. [42] for use in information retrieval, and returns a value $T(o, q) \in [0, 1]$ corresponding to the cosine of an angle between the query and document in a document keyword similarity space. For a given term τ and document p, the term frequency inverse document frequency (tf-idf) coordinate $\mathit{tfidf}(\tau, p)$ in the similarity space is computed as

$$\mathit{tfidf}(\tau, p) = \frac{f(\tau, p)}{|p.W|} \log \frac{|O|}{|\{\forall o \in O : \tau \in o\}|}, \tag{4}$$

where $f(\tau, p)$ is the number of times term τ appears in document p, $|p.W|$ is the number of terms in p, and $|O|$ is the size of the document set O. The $\log$ term on the right of equation 4 is the inverse document frequency; its denominator is the number of documents in O in which term τ appears. Note how parameter p can be either the query document q or a document $o \in O$.

3.1.3 Dynamic Ranking

We define a decay parameter $H(\Delta t) \in [0, 1]$ as follows:

$$H(\Delta T) = e^{-\lambda \Delta t}, \tag{5}$$

where $\Delta t = |q.t - o.t_o|$ is the absolute value of the difference between the query time $q.t$ and the time t_o the document o was defined. Decay constant λ is related to a constant half-life $t_{1/2}$ as follows:

$$\lambda = \frac{\ln(2)}{t_{1/2}} \tag{6}$$

Half-life $t_{1/2}$ defines the time period during which the textual match score $T(o, q)$ decreases by half its weighted value. For example, with $t_{1/2} = 1$ (e.g. one day), the value of the decay parameter $H(1)$ is 0.5 compared to $H(0) = 1$. For documents two days older than the query time $q.t$, $H(2) = 0.25$.

The score $R(o, q)$ of a document $o \in O$ defined at time t_o matching a query q at time $q.t$ is

$$R(o, q) = \alpha(1 - S(o.l, q.l)) + \frac{1}{H(\Delta t)}(1 - \alpha)(1 - T(o, q)), \tag{7}$$

where $\alpha \in [0, 1]$ is the relative weight assigned to the spatial match score. Ranking the scores $R(o, q)$ by sorting from smallest (rank 1) to largest, and choosing the first k in the ranked document list defines our ranking scheme. We call this scheme DRTF-IDF (dynamically ranked term frequency inverse document frequency) as it adapts to a continuous stream of new documents.

Other textual match models such as the language model [36] could also be used for $T(o, q)$ in equation 7 as long as their maximum value is 1. If $q.t = t_o$, then $H(\Delta t) = H(0) = 1$, and the textual match receives its full weight of $1 - \alpha$. Equation 7 provides the ranking process for our top-k spatial keyword query (TkSKQ) processing Algorithm 6.

3.2 Example

In this section, we illustrate top-k search based on dynamic ranking. The sample dataset consists of 14 spatially referenced objects, shown in Table 1. Note that the table displays the objects sorted in reverse chronological order, with the most recent document shown first.

Document	Text	Date	Distance(m)
14	unbelievable lobster taco	06/29/20	194
13	best T-bone steak	06/28/20	450
12	delicious lobster roll	06/21/20	250
11	nice steak	06/21/20	694
10	great shrimp and steak	06/17/20	294
9	smoked salmon waffle	06/14/20	219
8	nice Hawaiian pizza	06/14/20	581
7	excellent fajitas	06/12/20	262
6	very nice red snapper	06/12/20	400
5	great Vietnamese	06/05/20	394
4	best grilled steak	06/03/20	450
3	superb steak	05/30/20	294
2	very good enchiladas	05/28/20	575
1	best chimichangas ever	05/16/20	469

Table 1: Spatially referenced objects in Figure 1

An example query with query criteria $q(\{\text{best steak}\}, 06/30/20)$, and with query location at the center of the search disk shown in Figure 1, involves a query search radius of 500 m.

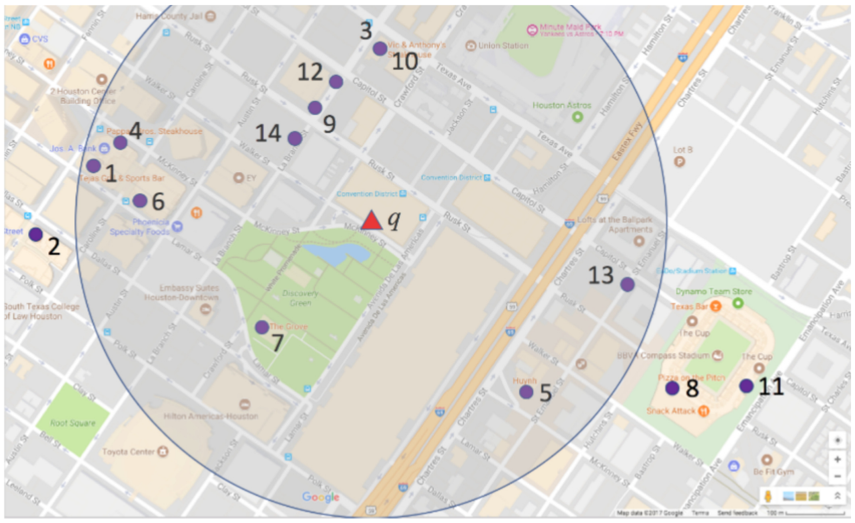


Figure 1: Example top-k search based on dynamic ranking.

Rank	Document	Text	$q.t - t_o$ (days)	Score
1	13	best T-bone steak	2	0.522
2	4	best grilled steak	27	0.623
3	11	nice steak	9	0.731
4	10	great shrimp and steak	13	0.887
5	3	superb steak	31	0.948

Table 2: Top-5 results for documents in Table 1, with $\alpha = 0.2$, half-life $t_{1/2} = 64$ days, query $q(\{\text{best steak}\}, 06/30/20)$

The outcome of executing the example query are the top-5 ranked results as shown in Table 2. As can be seen, even though documents 13 and 4 are the same distance from the query location, document 13 (2 days old) gets a lower score (resulting in a higher rank) compared to document 4 (27 days old) due to the dynamic ranking of equation 7. Document 14 is not included in the top-5 ranked results even though it is the closest to the query location, and the most recent. Equation 1 requires that $|q.W \cap x.W| > 0$, and this condition is not met for document 14.

Figure 2 demonstrates the dynamic effect of our temporally relevant ranking scheme on document 3 using four different query dates (31-May, 10-Jun, 20-Jun, 30-Jun), each issued with the same query words $q.W$ as in Table 2, with four half-lives $t_{1/2} \in \{8, 16, 32, 64\}$ days.

For Figure 2, we assume that the query time $q.t$ always exceeds the latest t_o for any document $o \in O$. The size of the document set $|O|$ changes over time on the four query dates, so document 3 competes with better scoring documents as $|O|$ increases. Notice how

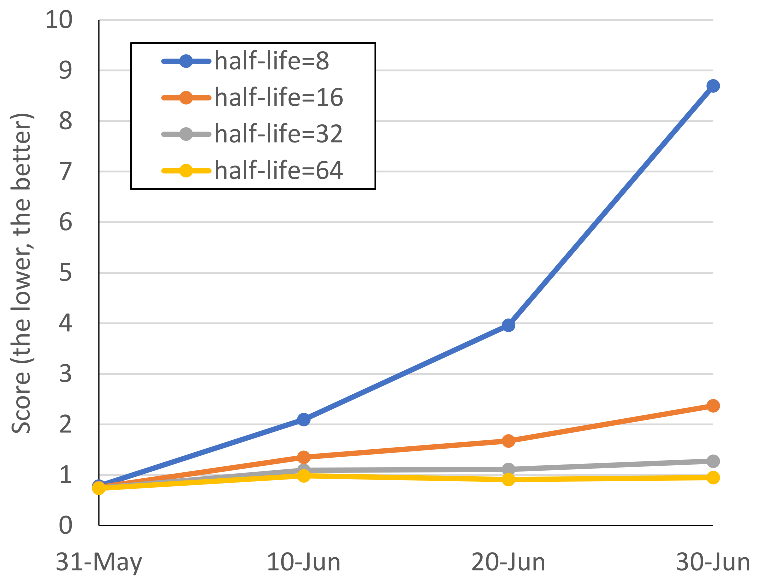


Figure 2: Document 3 score (Table 1) for different half-life $t_{1/2}$ values; lower scores give better ranks.

a half-life $t_{1/2}$ of 8 days means that document 3 (defined on 30-May) is 31 days old on 30-Jun, or almost four complete half-lives. The decay parameter $H(\Delta t)$ = H(31) = 0.068, which increases the weighted textual match score in equation 7 by a factor of $1/0.068 \approx 14$, and increasing its rank to 13.

4 Problem statement: spatio-temporal textual top-k query

Definition 2. *A spatio-temporal textual top-k query (TkSTTQ) returns the set Y of the k highest ranked documents in O that satisfy a query q, as follows:*

$$Y \equiv \text{sort}(\{\forall y \in Y : |q.W \cap y.W| > 0, \\ d(y.l, q.l) \leq cr, q.t_L \leq y.t_o \leq q.t_U, |Y| \leq k\}). \tag{8}$$

The query $q = (l, W, T, k)$ specifies a query location $q.l$, a query keyword set $q.W$, a query time range $q.T$ (where $T = \{t_L, t_U\}$ defines the time range lower bound t_L and upper bound t_U), and a parameter k where k is the number of ranked documents to return. Parameters c, r and function $d(y.l, q.l)$ are as defined in section 3. To assign scores and order the documents in Y, we use the following spatio-textual temporal combined score $C(y, q)$:

$$C(y,q) = \alpha(1 - S(y,q)) + \eta M(y,q) + \zeta(1 - T(y,q)) \\ \text{where, } \alpha, \eta, \zeta \in [0 \,..\, 1] : \alpha + \eta + \zeta = 1 \tag{9}$$

The spatial score $S(y, q)$ is computed using Equation 2. The textual score $T(y, q)$ is computed using Equation 3. The temporal score $M(y, q)$ is calculated based on the recency of the document time $o.t_o$ within the query time range $q.T$, and is computed as:

$$M(y, q) = 1 - \frac{o.t_o - q.t_L}{q.t_U - q.t_L} \tag{10}$$

Note that a lower temporal score results in a higher rank for the document, as with the TkSKQ search defined in section 3. Y is thus 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 8. We use a priority queue to maintain the sorted set Y of the (up to) k highest ranked documents while searching.

The textual score in TkSTTQ does not decay over time, which is the case in TkSKQ. Thus the ranking scheme of TkSTTQ is not dynamically updated. On the other hand, its temporal score component takes recency of the document into account and hence the combined ranking is still temporally relevant.

5 Pastri spatio-textual index

We present the system organization and the internal data structures of our spatio-textual index Pastri in this section. We also outline the update and query processing algorithms in detail.

5.1 System organization

To support high update throughput and low latency query processing, an effective use of CPU and memory resources should be made, while minimizing I/O. Our system follows a multi-threaded organization and utilizes an in-memory index. In Figure 3 we show the update and query processing workflows. Pastri is designed to be able to ingest document streams efficiently. As shown in Figure 3, upon receiving a geo-tagged document, such as a tweet, $o = (id, l, W, t_o)$, it is enqueued by our system into the queue RQ_t. Threads in the thread pool TP_t are known as *table insert threads* and one of them retrieves the newly received document from RQ_t. To store the document in our system a *DocumentTable* record is created based on the received document. A unique record id, *RID* is generated by incrementing a variable with an atomic operation. This unique id is produced when the record is inserted into the table *DocumentTable*. The schema of *DocumentTable* is as follows: {*DocumentId*,*Datestamp*,*Latitude*,*Longitude*,*Content*}.

Here, the column *DocumentId* is a unique identifier of the document and corresponds to the document field $o.id$. The column *Datestamp* is the time, t_o, representing the time when the document was created. The table columns *Latitude* and *Longitude* correspond to the document fields $o.l.lat$ and $o.l.lon$ and represent the location of the document, Finally, the *Content* column stores the textual content of the document $o.W$. Note that if the content $o.W$ is a small document, such as, a Facebook post, a tweet or blog comment, it can be stored directly in the *Content* column. On the other hand, if the content is too large to be stored in the *Content* column of the table, it can store the pointer to its location in the disk file. In our system, we encode the document text using dictionary encoding and store them as a list in the *Content* column of the *DocumentTable*.

The table, *DocumentTable*, is persisted into the stable storage by an update efficient data store LS_t. We implemented this data store by extending the LSM-tree, which involves maintaining multiple LSM-tree based partitions. Our extension, called pLSM store, can support

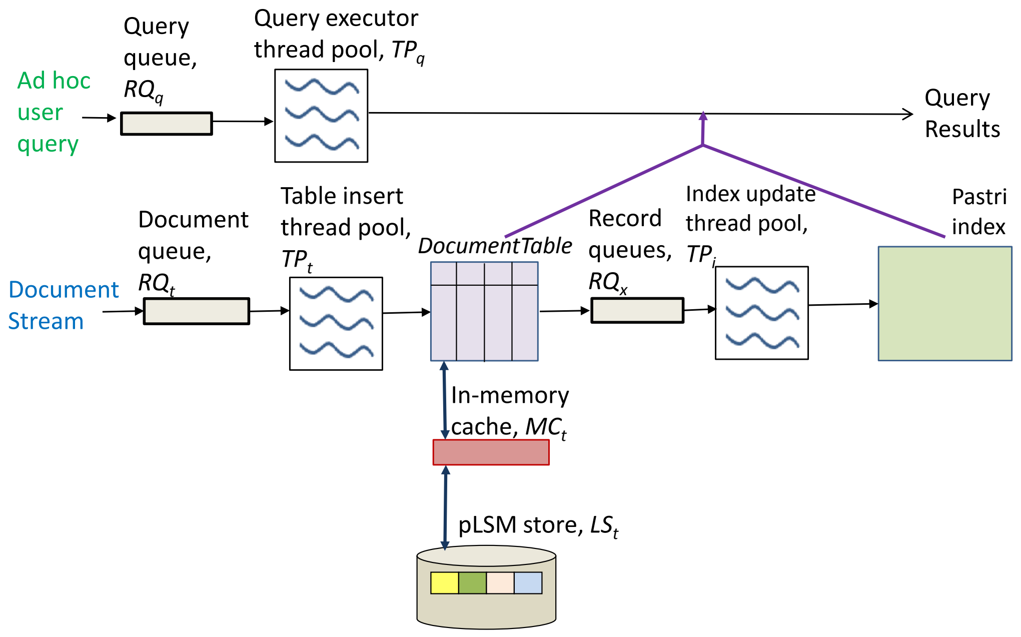


Figure 3: System architecture.

high data update rate that is suitable for ingesting document streams. Further details of our data store is provided in Section 5.3. To enable low latency operations an important consideration is how to minimize the disk I/O. To that end, our persistent data store is supported by an in-memory cache MC_t. The purpose of this cache is to maintain the most recent documents in memory and avoid disk I/O while accessing the records corresponding to such documents. Hence, when a document record is inserted into *DocumentTable*, it actually entails first inserting the record into MC_t and then into LS_t The in-memory cache MC_t can store a certain number of records based on the available system memory. Records from MC_t can be evicted based on different policies, for example, if the age of a record exceeds a configured threshold.

The next step of processing a newly received document is to update the index Pastri. After a document record is successfully created within *DocumentTable*, a copy of the record is enqueued in one of the queues in RQ_x. A thread pool TP_i is utilized to handle the entries in RQ_x. A thread from TP_i, called *index update thread*, is responsible for dequeuing the record from RQ_x and then updating the index. More details about the update processing algorithm is presented in Section 5.4. Further discussion about the inner working of the index is in Section 5.2.

As new query processing tasks are submitted, they are enqueued in a queue RQ_q. To support simultaneous processing of multiple queries, a thread pool TP_q is utilized. A thread in TP_q is called *query thread* and it is responsible for the next query task in RQ_q. A query processing task involves both the document table and the index. The details of the query processing algorithm is provided in section 5.5.

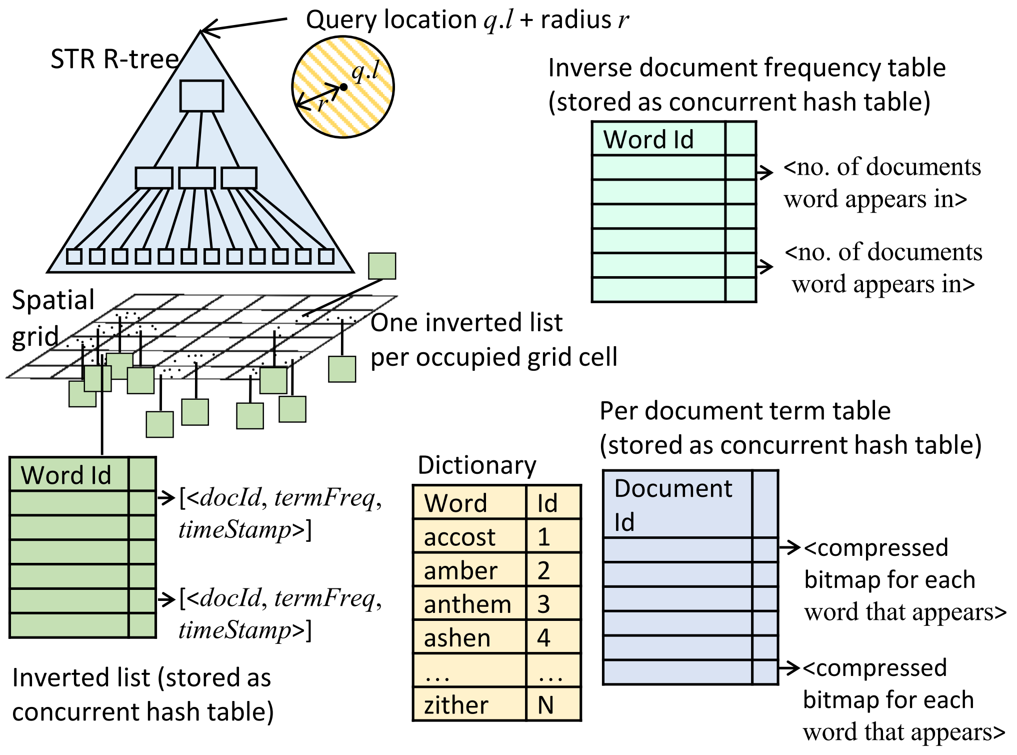

Query location $q.l$ + radius r
STR R-tree
$q.l$
r
Inverse document frequency table (stored as concurrent hash table)
Word Id
<no. of documents word appears in>
<no. of documents word appears in>
Spatial grid
One inverted list per occupied grid cell
Per document term table (stored as concurrent hash table)
Word Id
[<docId, termFreq, timeStamp>]
[<docId, termFreq, timeStamp>]
Dictionary
Word | Id
accost | 1
amber | 2
anthem | 3
ashen | 4
… | …
zither | N
Document Id
<compressed bitmap for each word that appears>
<compressed bitmap for each word that appears>
Inverted list (stored as concurrent hash table)

Figure 4: Structure of Pastri index.

5.2 Index structure

The spatio-textual index, Pastri, is an important aspect of our system. As shown in Figure 4, Pastri has several internal data structure components. It can be treated as a hybrid between a tree and a grid based index. The entire spatial region covered by the index is organized into a grid, *GRID*, with equal size regular grid cells. To index the cells of *GRID*, *STR*, a Sort-tile-recursive R-tree is utilized. This does a better job of packing the grid cells tightly than a regular R-tree. This is important in order to perform an efficient spatial search. We implement a circular range search algorithm with *STR* that returns the cells in *GRID*, which are within radius r from a location $q.l$. Note, *STR* is solely used for query processing and it is not involved in the document update.

To support textual indexing, inverted list based data structures are used. Specifically, an inverted list (*ILIST*) is maintained for each grid cell in *GRID*. This (*ILIST*) keeps track of the documents that are indexed at its corresponding cell. *ILIST* is implemented as a concurrent hashtable, where the key is a word id and value is a list of tuples. Each value tuple is comprised of 3 elements: the document id, the *term frequency*, and the timestamp when the document was generated. The term frequency represents the number of times a word appears in a particular document. The key of the hashtable (i.e. word id) corresponds to the numeric identifier in a global word dictionary (*DICT*). The dictionary, *DICT*, is a global data structure that serves to support dictionary encoding [1] of the text content in the documents and thus assists in data compression. Each word in the text of a new document is checked against *DICT*, and if a match is found, the word is replaced by its position in *DICT*. If no match is found, a new entry is created in the dictionary with the word as the key and its position as the value.

Apart from *DICT*, there are three global data structures. They include, an inverse document frequency table (*IDFT*), a per document term table (*PDTT*) and a global document counter (*DC*). Whenever a new document object u is received by our system, the corresponding cell within *GRID* is determined from its location (i.e. $o.l.lat$, $o.l.lon$), and then the counter *DC* is incremented. The *IDFT* data structure keeps track of inverse document frequency for each word, which is based on the number of unique documents in which a word appears. To represent each document concisely, a concurrent hash table data structure, *PDTT*, is maintained. In this data structure the key is a document id and value is a compressed bitmap (*CBMAP*). A bit in this bitmap is set if a keyword is present in the document, where the bit position corresponds to the dictionary encoded value of that word. To be memory efficient, *CBMAP* is compressed using a bitmap compression technique.

5.3 Storage structure

In this section we discuss our persistent data store. Since supporting high velocity streaming updates and low latency query processing are key goals of our system, minimizing disk I/Os is an important consideration. Recall that *DocumentTable* is backed by a disk-resident persistent data store.

The log-structured merge-tree (or LSM-tree) [35] is an insert efficient data structure, which supports key-value pair based indexed access to data files. An LSM-tree is particularly suitable for applications with high insert volumes, such as append-only logs. It consists of a memory-resident component to absorb inserts and defer disk writes. When the insert segments exceed a threshold, the in-memory component is merged into a corresponding disk component. A known drawback of the LSM-tree is that its multi-threaded scalability is not good. To illustrate this, we conducted an experiment with a persistent store based on a single LSM-tree and observe its insert throughput scalability by varying the number of insert threads. We used a synthetic dataset consisting of 1 million records, with the record size 100 bytes. As shown in Figure 5, the highest throughput was achieved with 1 insert thread. Increasing the number of threads from 1 to 2 threads caused a significant drop in throughput, which did not improve when the number of threads were 4, 6 or 8.

To improve the multi-threaded scalability of LSM-tree with regards to data ingestion, we introduce an enhancement to the regular LSM-tree based data store. In our persistent storage (shown in Figure 3), there are multiple LSM-trees $\{P_i | 1 \leq i \leq p\}$, each P_i is associated with a dedicated queue and an insert thread. We call this a *partitioned LSM-tree* store or *pLSM* store and it offers the same APIs for reading and writing data as an LSM-tree. While inserting a new document object, $o \in O$, the pLSM store uses an internal mapping $f : o \mapsto i$, which associates each object to a partition. Each o is first inserted into a queue corresponding to i. A dedicated insert thread then retrieves o from its queue and attempts to inserts it into partition i. We demonstrate in section 7.2.3 that the insert throughput of our persistent store scales well with the number of threads.

Note that LSM-tree has been previously used in systems, such as AsterixDB [9], for data storage. AsterixDB is a parallel database operating in a shared-nothing distributed cluster setting. It supports high-throughput data ingestion through partitioned parallelism [22]. In

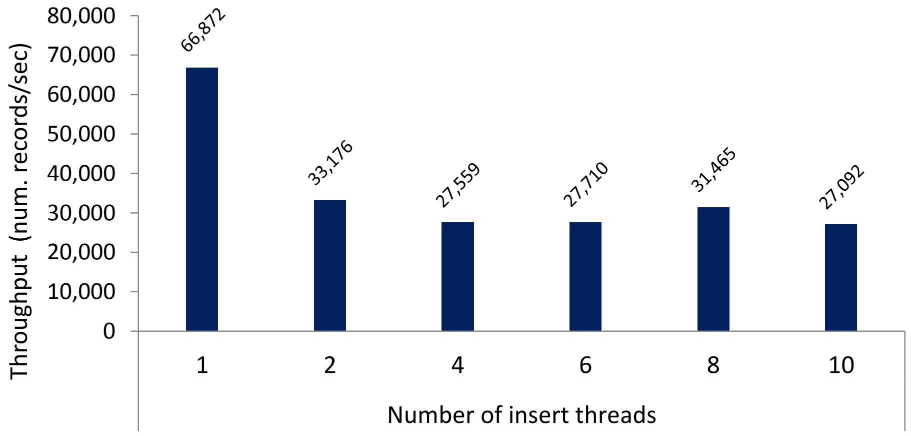


Figure 5: Scalability of LSM-tree.

Algorithm 1: Algorithm InsertDocumentTable

```
Input: $o$ is a document object to be inserted into the table, DocumentTable. It is backed
       by disk-based pLSMStore and an in-memory table memtable
Output: record id, rid, of the newly inserted document object
1 rid ← ridGen.atomicIncrement() ;
2 o.id ← rid ;
3 memtable.insert(o.id,o) ;
4 byteArrO ← serialize(o);
5 pLSMStore.put(o.id,byteArrO) ;
```

contrast, our approach Pastri is targeted for a single-node system and hence its data ingestion is focused on multi-threaded parallelism. In summary, Pastri's data ingestion mechanism is more fine-grained, whereas AsterixDB's data ingestion system is coarse-grained.

5.4 Update processing

In this section, we describe the update processing algorithm. It is designed to handle the ingestion of streaming geo-tagged documents. As a new document object, $o \in O$, arrives at our system, it is at first enqueued into RQ_t (in Figure 3) and retrieved by a thread of the thread pool TP_t. The threads in TP_t are called "table insert threads". Before a new record for the document can be inserted into *DocumentTable*, a new unique record id (*RID*) is generated. Then the new record is inserted. Algorithm 1 shows the steps to insert o into *DocumentTable*. First a unique *RID* is computed by an atomic operation and the object is inserted into an in-memory table. Then the object o is serialized and stored in the persistent pLSM store (lines 5 to 6).

Then an entry corresponding to the document record is enqueued into RQ_x (in Figure 3). The threads in the thread pool TP_i are known as "index insert threads" and they are utilized to handle the elements in RQ_x. One of these threads in TP_i retrieves the new entry from RQ_x and processes it by updating the data structures in the Pastri index. The steps involved in this process are shown in Algorithm 2. At first, the grid cell to which the

```
Algorithm 2: Algorithm ProcessIndexUpdate
   Input: o is a document object and ILIST is the inverted list of the grid cell in which o.l
          belongs to.
 1 Initialize term frequency hash table termFreqHM;
 2 for term in o.W do
 3     if term exists in termFreqHM.keys() then
 4         value ← termFreqHM.get(term) ;
 5         termFreqHM.put(term, value+1) ;
 6     else
 7         termFreqHM.put(term, 1) ;
 8 for term in termFreqHM.keys() do
 9     termFreq ← termFreqHM.get(term) ;
10     if term exists in ILIST.keys() then
11         invList ← ILIST.get(term) ;
12     else
13         Instantiate a new invList ;
14     invListEntry ← (o.id, termFreq, $t_0$) ;
15     Add invListEntry to invList;
16     ILIST.put(term, invListEntry);
17     IDFT.update(term);
18 PDTT.update(o);
19 DC.increment();
```

```
Algorithm 3: Algorithm Round-robin
   Input: o is a document object, QueueArray is an array of queues, and NUM_WORKER
          is number of index insert threads. Atomic integer nxtQId is global.
   /* Table insert thread                                                      */
   QueueArray[nxtQId.increment() % NUM_WORKER].add(o);
   /* Index insert thread:  id localThId                                       */
 1 Initialize: LocalQueue ← QueueArray[localThId] ;
 2 while true do
 3     o ← LocalQueue.pop() ;
 4     Process o and insert into index ;
```

new document belongs to needs to determined. This is done with the location field *o.l* of the corresponding record. As each grid cell has an associated inverted list *ILIST*, the next step is to update it by registering a new entry in *ILIST* for every word in that document (lines 8 to 16). The last step is to update the global data structures *IDFT*, *PDTT* and *DC* (lines 17 to 19).

Algorithm 4: Algorithm Work-stealing

```
  Input: Same as in Algo. 3. Additionally, rnd is random number generator.
  /* Table insert thread                                              */
  QueueArray[rnd.nextInt(%NUM_WORKER)].add(o);
  /* Index insert thread:  id localThId                               */
1 for $nxt \leftarrow 0$ to $QueueArray.len() - 1$ do
2     o ← QueueArray[nxt].pop() ;
3     Process o and insert into index ;
```

Algorithm 5: Algorithm Affinity

```
  Input: Same as in Algo. 3 . Additionally, Gcell2ThreadMapping is a hashtable that is
         populated by an offline algorithm AffinityAssignment, which maps grid cells
         to index insert threads.
  /* Table insert thread                                              */
1 cellId ← o.l;
2 QueueArray[Gcell2ThreadMapping.get(cellId)].add(o);
  /* Index insert thread:  id localThId                               */
  Same as steps 1 to 4 in Algo. 3 .
```

5.4.1 Load-balancing and handling skew

The sources of data streams, such as micro-blogs, are not evenly distributed. This is dependent on the geographical location and it can be expected that more data is generated from regions where the population density is higher. As a result, the source locations of the geo-tagged documents can be skewed. Such skew can have a significant effect on the update performance of a grid-based index. Update processing threads dealing with dense regions will perform more work than the ones processing sparse regions. With high update rates, load-balancing becomes quite important. The objective of such a load-balancing algorithm is to equally distribute the work involved in update processing among the update processing threads. If this is achieved, then the variation in the number of geo-tagged documents that are processed by these threads will be minimized. In practice, equal allocation of tasks among workers can be challenging. To explore the trade-offs, we propose 3 load-balancing algorithms: Round-robin, Work-stealing, and Affinity. We discuss them next.

Round-robin. The steps in Round-robin are presented in Algorithm 3. In this approach, a table insert thread from TP_t after having inserted the next document record into *DocumentTable*, then proceeds to insert the record into a queue in RQ_x in a round-robin manner. Recall from Figure 3 that RQ_x represents an array of queues. In Round-robin approach each queue in RQ_x is associated with a thread in the thread pool TP_i. Therefore, the selection of next queue from RQ_x in a round-robin way can be done by incrementing the atomic counter *nxtQId* and then performing a modulo operation (line 1 in Algorithm 3). Subsequently, the inserted record is picked up by the corresponding worker thread in TP_i (line 4), which then updates the index based on this record (line 5).

Work-stealing. The steps involved in this load balancing algorithm is shown in Algorithm 4. As the name suggests, the threads in TP_i perform work-stealing while retrieving

Algorithm 6: Algorithm ProcessQuery

```
Input: q is a given query and the goal is to find top k objects. Initial search radius r.
       Priority queue PQueue stores the matched results.
1  cellsSeen ← $\phi$ ;
2  for numAttempts=1 to MAX_ATMPTS do
3      distLim ← initOrUpdate(r, numAttempts) ;
4      cellsFound ← CircularRangeQry(q, distLim, cellsSeen) ;
5      cellsSeen ← cellsSeen ∪ cellsFound ;
6      for cell2process in cellsFound do
7          ILIST ← cell2process.invertedList() ;
8          Initialize perDocTextualScore ;
9          for term in q.W do
10             if term exists in ILIST.keys() then
11                 invList ← ILIST.get(term) ;
12                 for (o.id, termFreq, $t_0$) in invList do
13                     cbitmap ← PDTT.getbitmap(o.id) ;
14                     if matches(cbitmap,term) then
15                         textRelScore ← calculate textual relevance score per Equation 3 ;
16                         perDocTextualScore .update(o.id,txtRelScore);
17         for o in perDocTextualScore.keys() do
18             textRelScore ← perDocTextualScore.getScore(o) ;
19             spatialRelScore ← calculate spatial relevance score per Equation 2 ;
20             cScore ← calculate combined relevance score per Equation 7 ;
21             sIndx ← k;
22             if PQueue.size() < k then sIndx ← PQueue.size() ;
23             idxScore ← PQueue.getScoreAt(sIndx-1) ;
24             if compare(cScore,idxScore) then
25                 PQueue.add(o.id,cScore) ;
26     if PQueue.size() ≥ k then break ;
```

records from RQ_x. To achieve this, unlike in Round-robin, the threads in TP_i are not associated to any queue in RQ_x in this approach. Instead, each thread iterates over the array of queues in RQ_x (line 3 in Algorithm 4). If it finds a new record to be processed in the queue currently being inspected, it is processed (lines 3–4). After that it moves on to check the next queue.

Affinity. The Affinity load balancing algorithm is shown in (Algorithm 5). The main idea behind this approach is for an index insert thread to process records destined for a specific set of grid cells in the Pastri index (Figure 4), rather than any grid cell. Since some grid cells cover areas that are denser than the others, with this approach workload distribution can explicitly consider the object density. Specifically, a mapping of grid cells to index insert threads, *Gcell2ThreadMapping*, is produced by an offline algorithm *AffinityAssignment* (skipped due to space constraint), such that the standard deviation of the object densities in the grid cells assigned to the threads is less than a threshold. To determine per cell object density, *AffinityAssignment* periodically performs a density sampling of each cell by calculating the number of documents originating from a grid cell in previous iterations. Then an index insert thread from the TP_i just processes those records that belong to the grid cells as determined by *Gcell2ThreadMapping*.

In Section 7.2.1, we evaluate the above-mentioned load-balancing algorithms.

5.5 Query processing

In this section we describe the algorithms involving top-k query processing. Our system supports both temporally relevant top-k spatial keyword query (TkSKQ) and spatio-temporal textual top-k query (TkSTTQ). We mainly describe the processing of TkSKQ, since the execution of TkSTTQ follows a very similar procedure.

Our system has been designed to enable efficient execution of TkSKQ over dynamically ranked document streams. Algorithm 6 outlines a partial listing of the query processing algorithm. For a TkSKQ query instance q, Algorithm 7 shows how to perform a circular range search, in which the point of origin is $q.l$ and the initial search radius r. Note that r is configurable. This search process involves traversing the STR R-tree component of the Pastri index (Figure 4) to find all grid cells that are within distance r from $q.l$. A priority queue *LocalPriQueue* is used to keep track of the found cells. If a node of the STR R-tree is a leaf node that is within the desired distance, then it is included in the set of found cells *cellsFound* (lines 6–8 in Algorithm 7). Otherwise, it is an internal node and so its children must to checked (lines 10–12 in Algorithm 7).

For every cell that is in the list of cells returned by Algorithm 7, the actual document records are processed and their textual and spatial relevance scores are calculated. To that end, at first the corresponding inverted list *ILIST* component corresponding to the particular cell is obtained (line 7 in Algorithm 6). Then the textual relevance score is calculated for each document that contains one or more query keywords (lines 8–16 in Algorithm 6). Subsequently, for each of those documents the spatial relevance score and the combined score is calculated (lines 18–20 in Algorithm 6). A priority queue $PQueue$ is used to maintain the current top k documents. If the calculated combined score of a document is less than the score of the k-th document in $PQueue$, it is enqueued (lines 21–25 in Algorithm 6).

In Figure 6, the processing of TkSKQ is visually illustrated with two queries: *Q1* and *Q2*. The purple dots correspond to location of the documents. When a query q is issued, a list of grid cells that are within distance r from $q.l$ are returned by Algorithm 7. The red circles represent the query radius. Then, all the documents in these cells are processed by Algorithm 6. For query *Q1*, the light blue shaded cells are covered by the initial query radius r. If k matches are found, Algorithm 6 returns them as the search result. Otherwise, the search radius must be enlarged and the process is repeated for each expansion of the radius. For query *Q2* this expansion was required until k matching documents were found. The pink shaded cells are all the cells that were processed during the execution of query *Q2*. Since this process of expanding of the search radius and processing the covered cells can continue indefinitely, a maximum attempt threshold *MAX_ATMPTS* is used. If the maximum attempt threshold *MAX_ATMPTS* is exceeded, search terminates and query returns with the matching documents found so far (if any).

Algorithm 7: Algorithm CircularRangeQry

```
Input: q is a given query, distLim the search radius and cellsSeen are cells processed in
       prev. iterations. STR is the STR R-tree that indexes the grid cells.
1  cellsFound ← $\phi$ ;
2  node ← STR.getRoot() ;
3  LocalPriQueue.add(node) ;
4  while LocalPriQueue.hasMoreElts() do
5      node ← LocalPriQueue.poll() ;
6      currDist ← distance(node,q.l) ;
7      if currDist $\geq$ distLim then break ;
8      if node.isLeaf() then cellsFound ← cellsFound $\cup$ node ;
9      else
10         for cnode in node.children() do
11             if distance(cnode,q.l) $<$ distLim then
12                 LocalPriQueue.add(cnode) ;
13 return cellsFound $\setminus$ cellsSeen
```

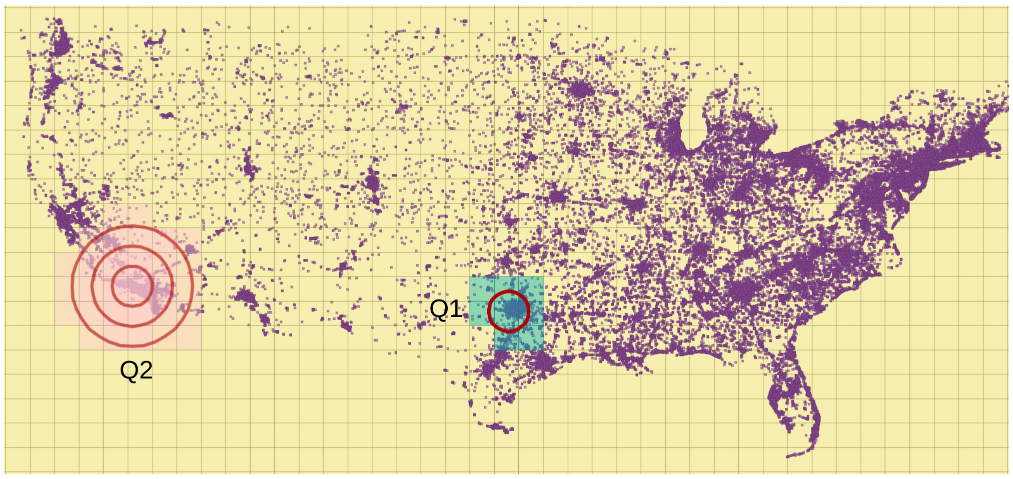


Figure 6: Illustration of query processing in Pastri.

6 Analysis

In this section we provide detailed cost analysis of the insert and query processing algorithms. Specifically, we prove a tight bound for the cost of inserting N data items into a single LSM tree, which leads to a tight bound on the amortized cost to insert a single data item into a pLSM store. An upper bound on the cost of retrieval (query) is also provided.

Algorithm 8: LSMinsert($\mathcal{C}, C_0$)

```
Input: The compaction set implemented as a stack $\mathcal{C}$
       The main memory component $C_0$
Output: Updated compaction set stack $\mathcal{C}$
1 merged ← false ;
2 $c \leftarrow$ describe($C_0$) ;
3 while $|c| \geq |\text{top}(\mathcal{C})|$ do
4     $c \leftarrow$ merge($c$, pop($\mathcal{C}$)) ;
5     merged ← true ;
6 if not merged then
7     write($c$) ;
8 push($\mathcal{C}$, $c$ ) ;
```

6.1 Insertion

The running time in I/Os of the **InsertDocumentTable** Algorithm 1 depends on the cost to insert records into the *DocumentTable* at line 5. The *DocumentTable* is stored in a pLSM (partitioned log-structured merge-tree) store (see e.g. [35]). In our case, the pLSM store is organized as P LSM-trees, where P is the number of LSM-trees comprising the pLSM store. As keys are inserted into a pLSM store (line 5 in Algorithm 1), it internally distributes them equally among the P LSM-trees.

Each LSM-tree contains hierarchical, sorted (by key) components $C_0, C_1, ..., C_k$, where C_0 is stored in main memory. The k remaining components all reside on non-volatile external memory such as rotating magnetic disks. As in [35], we assume that each successive hierarchical component C_i holds up to f times the number of items held by C_{i-1}; i.e.

$$|C_i| = f|C_{i-1}|, \forall i \in \{1, ..., k\} \tag{11}$$

where C_0 can hold up to M items, or $\frac{M}{B}$ blocks of data. We follow the model of Aggarwal and Vitter [3] [49] where a block of data transferred to external memory holds B data items, internal memory can hold at most M data items, and the entire tree stores N data items. The LSM-tree merges components of similar size, which means it does not merge components with every write of $\frac{M}{B}$ data items to external memory. The LSM-tree insertion algorithm is adapted from Scully [43], as described in Algorithm 8.

The compaction set $\mathcal{C}$ and c both store data describing the components (i.e. their size and starting locations in memory). The describe(C_0) function at line 2 returns the description of C_0. Algorithm 8 assumes that the size of the top entry of an empty stack returns 0. Merging of similar sized components is carried out at line 4, and involves reading and writing blocks of the two memory components being merged. The merged component is then pushed back on top of the stack. If no merge occurred (if $|c| < |\text{top}(\mathcal{C})|$), then the main memory contents are written to external memory at line 7. The merged component description c is pushed onto the compaction set stack $\mathcal{C}$ at line 8. Here, we have $f = 2$ in equation 11.

Theorem 6.1. *Using Algorithm 8, the cost $I(N)$ in I/Os to insert N data items into an LSM-tree is bounded as* $2\frac{N}{B}\log_2\frac{N}{B} - 4\frac{N}{B} + 5 \leq I(N) \leq 2\frac{N}{B}\log_2\frac{N}{B}$.

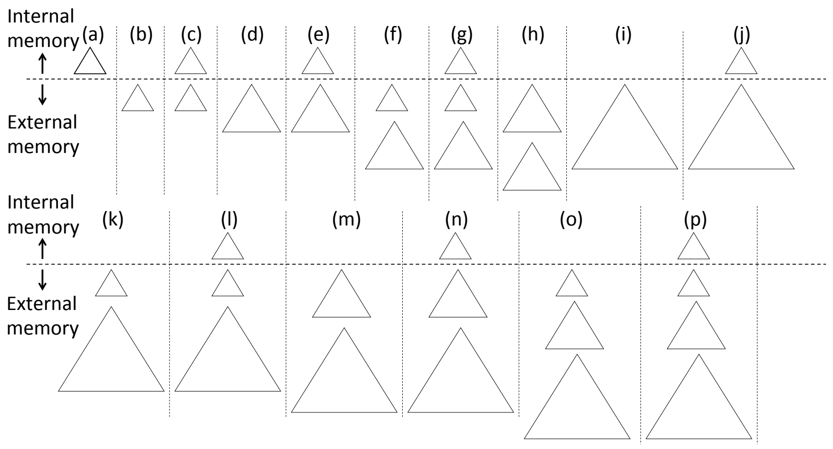


Figure 7: Progression of LSMinsert($\mathcal{C}, C_0$) algorithm I/O operations up to $8M$ data item inserts.

Proof. The number of I/Os required by Algorithm 8 is determined by the merge(c, pop($\mathcal{C}$)) operation at line 4 and the write(c) operation at line 7. Each merge(c, pop($\mathcal{C}$)) statement merges two already sorted components of similar size. Figure 7 illustrates the start of the process.

State (a) has $\frac{M}{B}$ blocks of data in main memory, and transforms to state (b) after $\frac{M}{B}$ blocks of data are written to external memory at line 7. At state (c), main memory contains another $\frac{M}{B}$ blocks of data, and the merge operation at line 4 reads $\frac{M}{B}$ blocks of data from external memory, and writes $2\frac{M}{B}$ blocks of data to external memory. At state (d) the cost is $(1+2+1)\frac{M}{B}$ = $4\frac{M}{B}$ I/Os. This pattern repeats up to state (h), incurring an additional $4\frac{M}{B}$ I/Os. At state (h), the top of the stack $\mathcal{C}$ points to the first set of $2M$ data items written to external memory indicated at state (d), and c points to the second set of $2M$ data items in external memory. These sets are both the same size, so a second merge operation is invoked at line 4, which costs $(2+2+4)\frac{M}{B}$ I/Os. At state (i), the total cost is $(2(1+2+1)+(2+2+4))\frac{M}{B} = 16\frac{M}{B}$ I/Os. The pattern repeats, as illustrated in states (j) to (p), resulting in the following recurrence relation:

$$T(n) = 2T(\frac{n}{2}) + 2n \tag{12}$$

with $T(1) = 0$, and where $n = \frac{N}{B}$. Solving this recurrence gives $T(n) = 2n\log_2 n$. This results in the upper bound of $2\frac{N}{B}\log_2\frac{N}{B}$ I/Os as claimed in Theorem 6.1.

The lower bound of Theorem 6.1 arises due to the delay in merging until as late as possible. Queries to find data items indexed by the LSM-tree are answered even if up to M data items are in internal memory. Thus, the minimum number of I/Os for insertion occurs when the greatest number of data items are in internal memory, and the next insertion forces a merge. States (a), (c), (g), and (p) in Figure 7 illustrate such "minimum I/O" states. State (a) incurs 0 I/Os, state (c) has incurred $\frac{M}{B}$ I/Os, state (g) (after inserting $4M$ data items) has incurred $5\frac{M}{B}$ I/Os and state (p) (after inserting $8M$ data items) incurs the $5\frac{M}{B}$ I/Os from state (c) plus the minimum cost of merging and writing another $4M$ data items, or another $16\frac{M}{B}$ I/Os. Extending the pattern in a similar fashion, we see that the "minimum I/O" cost for inserting $16M$ data items is an additional $48\frac{M}{B}$ I/Os. The recurrence relation

$$T(n) = T(\frac{n}{2}) + n\log_2 \frac{n}{2} \tag{13}$$

defines the number of I/Os, with $T(1) = 0, T(2) = 1$, where $n = \frac{N}{B}$. Solving this recurrence gives $T(n) = 2n\log_2 n - 4n + 5$. The lower bound of Theorem 6.1, i.e. $2\frac{N}{B}\log_2\frac{N}{B} - 4\frac{N}{B} + 5$ I/Os, is proven. □

Theorem 6.1 leads to the amortized cost of inserting a data item into an LSM-tree holding N data items as follows:

Corollary 6.1.1. *Using Algorithm 8, the amortized cost $I(N)$ to insert one item into an LSM-tree is $\Theta(\frac{log_2\frac{N}{B}}{B})$ I/Os.*

This result is the first we are aware of that gives a tight bound on the number of I/Os (amortized over N inserts) required to insert a data item into an LSM-tree. In addition, the partitioned LSM store used here reduces the insertion cost by a factor of $\frac{1}{P}$ as each key is inserted into only one of P LSM-trees. We arrive at the following amortized cost for the partitioned LSM store:

Corollary 6.1.2. *Using Algorithm 8, the amortized cost $I(N)$ to insert one data item into an LSM store consisting of P LSM-trees is $\Theta(\frac{log_2\frac{N}{BP}}{BP})$ I/Os.*

Proof. The proof is obtained by substituting $\frac{N}{P}$ for N in Theorem 6.1, and dividing by N to amortize the cost. □

This result assumes a single disk model [3] with sequential access. If the P LSM-trees can be stored on P external stores (e.g. disks as in the multi-disk model [50] [6]), this bound can be improved.

6.2 Query

The running time in I/Os of Algorithm 6 (**ProcessQuery**) depends on the cost to find documents matching the $q.W$ terms, and update the document textual score. In the worst case, the **for** loop at line 6 visits all the grid cells in *cellsFound*, and finds that all documents in each grid cell have terms that match one or more terms in $q.W$. In the normal case, we can assume that all operations of Algorithm **ProcessQuery** are done in main memory; i.e. all data is stored in the cache. In this case, the cost is zero I/Os. If cache is limited, one or more of the LSM-trees in the pLSM store might need to be visited. In this case, the cost to retrieve the document for each point found in range of the query of radius r centered at $q.l$ is $O(\log_B \frac{N}{P})$ I/Os [10]. This cost assumes that the LSM-trees are using Bloom filters to avoid unnecessary searches of external memory components.

7 Evaluation

We describe the experimental evaluation of our system in various settings in this section. First we outline the datasets and the settings and then we describe the experiments. We present a performance study of our system against two popular spatio-textual indexes, IR-tree and I^3. These two indexes do not support real time ingestion of document streams, however, both of them have support for ad hoc (snapshot) TkSKQ query execution.

A direct performance evaluation against systems that support document streams, such as Taghreed [26] and Mercury [27], cannot be made because they do not support ranked TkSKQ search. In Section 1 we presented a comparison of Pastri against Taghreed and Mercury, based on their reported ingestion throughput and query latency, which indicates that Pastri's data ingestion performance is significantly better.

7.1 Experimental setup

We present the experimental setup in this section. This includes, the details regarding the datasets, the query sets and setting of the experimental environment and key parameters.

7.1.1 Datasets

The performance evaluation was conducted with two categories of datasets. Further information regarding them are mentioned next.

Twitter datasets: These datasets consist of tweets that were geo-tagged [14]. The geo-tagging process involved using a real road network dataset [19]. Based on this dataset, the tweets from the original Twitter dataset were geo-tagged [14]. We created two additional datasets in order to perform scalability experiments. Overall, there are three datasets, namely, 200k, 2mi, and 20mi consisting of 200 thousand, 2 million, and 20 million tweets, respectively. Table 3 shows the details of these datasets. In order to simulate live streaming of the data, a driver program was implemented. This program stored the tweets into a queue from which they were retrieved for further processing by different system components.

Synthetic datasets: These datasets were generated in two steps. First, to generate the locations, we utilized the GSTD tool [47]. It can generate synthetic data for point or rectangular objects that follow one of several possible distributions. We used GSTD to generate point datasets for three different distributions: Random, Skewed, and Gaussian. In Figure 8 the pictorial representations of the scaled version of datasets are shown. In the second step, each location generated in the previous step is associated with a textual document. It has been observed that natural language text follows Zipf distribution. To select the text for a document, we developed an algorithm that generates words from a well-known English language corpus data [33] following Zipf distribution. A summary of the datasets and their properties are shown are in Table 4.

Wikipedia dataset: The georeferenced Wikipedia dataset [52], extracted from the official Wikipedia dump, has 280,000 articles with an average of 1018 words per article. The features of this dataset are summarized in Table 5.

7.1.2 Query set

A query workload was generated for each dataset. Each workload contains 1000 queries. Each query in a particular workload specifies a query location, a timestamp, and several keywords. During the query workload generation process a selectivity parameter is specified. As an example [34], if the selectivity is 5%, it means that there is a 5% chance that

Dataset name	Number of tuples	Average number of keywords	Maximum number of keywords	Average document length
200k	200,000	5.08	30	25.58
2mi	2,000,000	5.06	70	25.55
20mi	20,000,000	5.70	70	28.89

Table 3: Twitter datasets

Dataset name	Number of tuples	Average number of keywords	Maximum number of keywords	Average document length
Gaussian	1,000,000	8.5	16	41.45
Random	1,000,000	8.5	16	41.44
Skewed	1,000,000	8.5	16	41.44

Table 4: Synthetic datasets.

the query being generated, q, will contain all the keywords from an existing object o from the dataset when, $|q.W| \leq |o.W|$, or w keywords when, $w = |o.W|$ and $|q.W| > |o.W|$. Otherwise, there is a 95% chance that the query location will be randomly selected and the keywords will be chosen randomly based on the dictionary for that particular dataset.

7.1.3 Environment setup

The experiments were conducted on a machine having 16 AMD Opteron processing cores, each running at 2.8 GHz. The machine has 128 GB memory, and runs Red Hat Enterprise Linux 4.8 64-bit OS. The key parameters of the experimental settings are in Table 6.

7.2 Update throughput of Pastri

Next we evaluate the update throughput of Pastri in isolation. Particularly, we examine the effects of load-balancing algorithm, skew, and the number of partitions in pLSM store on update throughput.

7.2.1 Load-balancing algorithm

Load-balancing is necessary to deal with data skew. In these experiments the impact of the three load-balancing algorithms on the update throughput is evaluated. These algorithms are: Work-stealing (Algorithm 4), Affinity (Algorithm 5), and Round-robin (Algorithm 3). To isolate different contributing factors of the update throughput, the number of partitions

Dataset name	Number of tuples	Average number of keywords	Maximum number of keywords	Average document length
Wikipedia	280,000	390.14	7,770	1,018.39

Table 5: Wikipedia dataset.

Parameter	**Settings**
Grid resolution	64, 256, **1024**, 4096
Updates (num. documents)	200000, **2000000**, 20000000
Number of queries	**1000**, 1000000
k	5
Query radius	1 km, 10 km, **100 km**
Number of query keywords	1, 2, 3, 4, **5**
Query selectivity	1%, 5%, 10%, **20%**, 40%
Half-life, $t_{1/2}$	7 days

Table 6: Parameter settings (default parameters in bold).

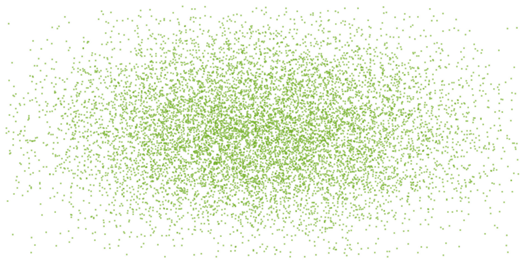
in pLSM store is fixed to 1. Figure 9 shows the observed update throughput with Twitter 20mi dataset as the number of index update threads was varied. As can be seen, the update throughput significantly improves when the number of threads increase from 1 to 2 with all three algorithms. It remains relatively stable with 4 threads. Beyond 4 threads, the update throughput either remains the same or is reduced. Among the 3 load-balancing algorithms, Algorithm Affinity achieved the best update throughput in all cases. With Algorithm Work-stealing, the update throughput significantly degrades with 8 index update threads due to contention.

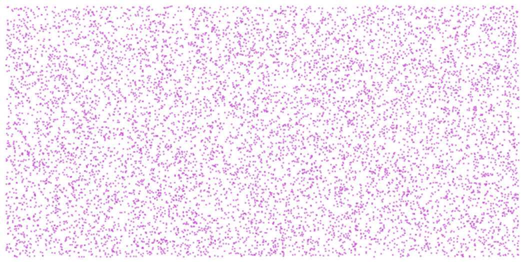
7.2.2 Grid resolution

Figure 10 shows the update throughput of Pastri for the three datasets as the grid resolution (number of grid cells) is varied to 64, 256, 1024, and 4096 cells. The dataset was Twitter 20mi and the number of partitions in pLSM store was fixed to 1 and the load-balancing algorithm was set to Affinity. As shown in Figure 10, the update throughput remains relatively stable irrespective of the grid resolution. The maximum and minimum difference between the best and worst update throughputs are only 6% and 2% respectively. These results indicate that Pastri's load-balancing approach does a reasonably good job of handling data skew.

7.2.3 pLSM store partitions

In Sections 7.2.1 and 7.2.2 we examined the impact of load-balancing algorithms and grid resolution on update throughput respectively, while keeping the number of partitions in pLSM store fixed at 1. In this section, we present the results of experiments in which the number of partitions in the pLSM store was varied, while the load-balancing algorithm was set to Affinity and the grid resolution was fixed at 1024. Figure 11 shows the update throughputs with the three Twitter datasets (200k, 2mi, and 20mi). The update throughput significantly improves when the number of partitions increases from 1 to 2 and then from 2 to 4. For instance, with dataset 2mi, the update throughput with 1, 2, and 4 partitions are 133k, 187k, and 206k objects/second respectively. Beyond 4 partitions, the update throughput does not improve, because it saturates the disk bandwidth. The throughput is the lowest with the smallest dataset 200k, since it does not saturate the processing capacity. The throughput with the largest dataset 20mi is higher than that of dataset 200k, but lower than that of dataset 2mi. This occurs due to the disk bandwidth becoming the limiting factor at high data volume.


(a) Gaussian


(b) Random

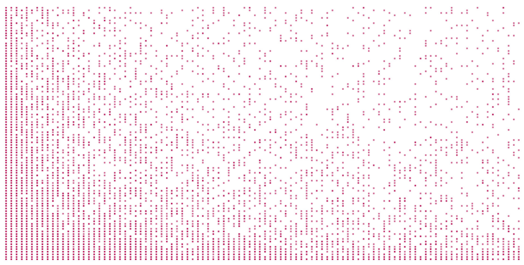

(c) Skewed

Figure 8: Distribution of the data points in each synthetic dataset (scaled).

7.3 Update performance comparison

We compare the update performance of Pastri against that of I^3 index [54] [1] and IR-tree [18] [1]. IR-tree is one of the first indexes to support efficient ad hoc TkSKQ. I^3 is a more recently proposed index. It significantly improves query performance over that of IR-tree. Unlike Pastri, I^3 and IR-tree do not support continuous updates, as they require the entire dataset a priori. So, instead of update throughput, we compare the the data loading and index

[1]We thank the authors for generously providing the code .

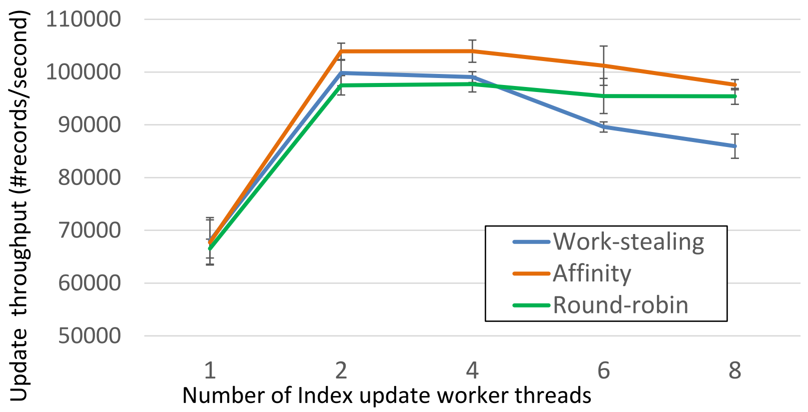


Figure 9: Update throughput of Pastri with different load-balancing algorithms (dataset 20mi).

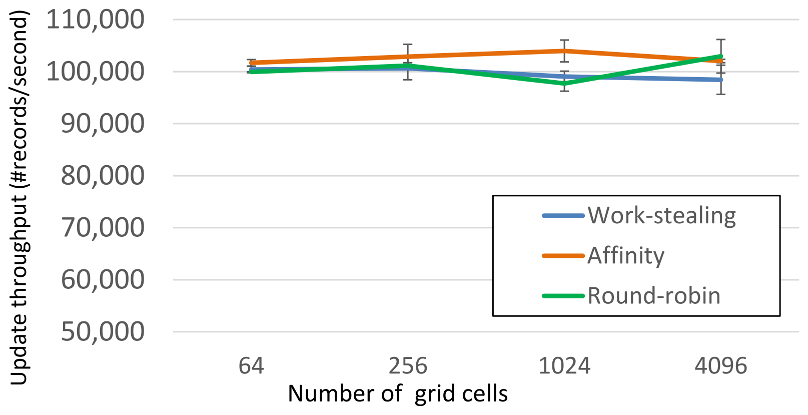


Figure 10: Update throughput of Pastri with different grid resolutions (dataset 20mi).

building time. For Pastri, the index building time is the total time to ingest the entire dataset.

7.3.1 Index building cost

In this section we evaluate the index construction cost. Specifically, the indexes IR-tree and I^3 were constructed with the three Twitter datasets (200k, 2mi, 20mi), as well as the synthetic datasets (Gaussian, Random, Skewed). The total data loading and index building time are reported for these indexes. Since Pastri is a streaming spatio-textual index, it does not require the entire dataset upfront and can ingest data and update index as new documents arrive. Hence, in the case of Pastri records were streamed using a driver program for each of the datasets. The time to completely ingest all the records and update the index is reported for Pastri. As shown in Figure 12, Pastri takes about 2 orders of magnitude less time than that of IR-tree and an order of magnitude than that of I^3 index. For instance, with dataset 2mi (Figure 12(a)), Pastri takes 10 seconds, I^3 needs 103 seconds and IR-tree takes

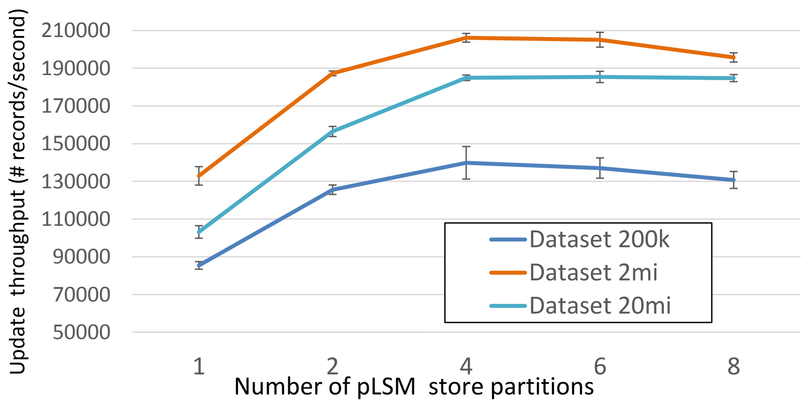


Figure 11: Update throughput of Pastri with different number of storage partitions.

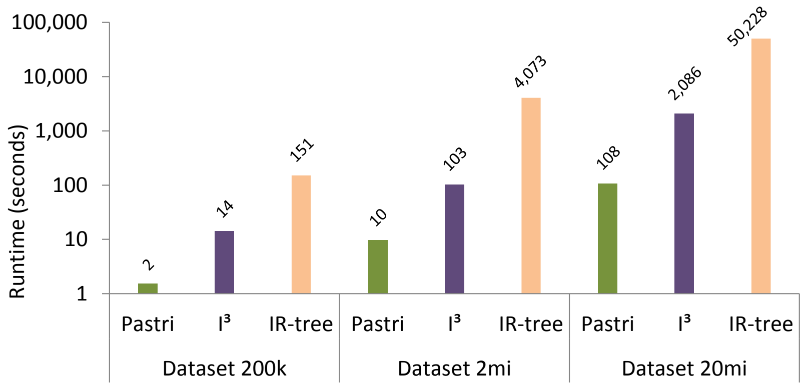
4,073 seconds. This trend continues with the Wikipedia dataset (see Table 5). This dataset is quite distinct, as it has significantly more words per document on average, compared to other datasets. As can be seen in Figure 12(c), with this dataset the index construction with Pastri is 7.7× faster than I^3 and 84.7× faster than IR-tree.

7.3.2 Storage cost

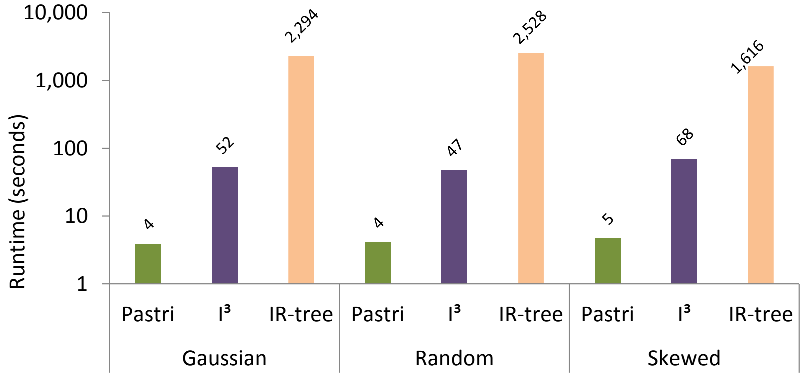
Both I^3 and IR-tree index use disk files to store both data and index. Pastri stores the data in the disk-resident pLSM store. It builds a main memory index from the data and does not actually store the index. The pLSM store uses compression, which significantly reduces disk storage cost. In Figure 13, we compare the disk usage (MB) of Pastri against that of IR-tree and I^3 with the Twitter datasets. As can be seen, the storage requirements of Pastri is significantly lower than the other two indexes. For instance, with the largest dataset 20mi, the storage cost of Pastri, I^3 and IR-tree are 1,114 MB, 30,161 MB, and 37,063 MB respectively. Results for synthetic datasets are omitted, as they show similar trends.

7.4 Parallel query throughput of Pastri

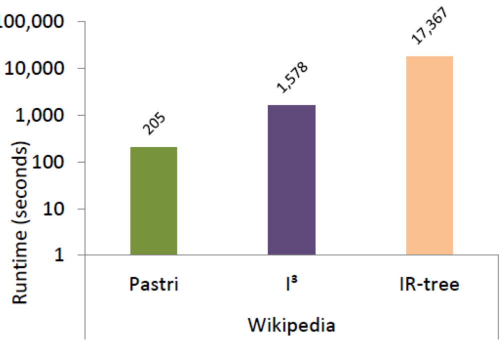
Since Pastri supports inter-query parallelism, here we demonstrate the multi-threaded query performance, in terms of query throughput (number of queries per second). For these experiments all parameters are set to their default Table 6 values except for the number of queries. One million TkSKQ queries were executed with Pastri for each of nine different query execution thread counts of 1, 2, 4, 6, 8, 10, 12, 14, and 16. The query throughputs achieved with the Twitter datasets are plotted in Figure 14 (in **log scale**). As can be seen in Figure 14, the query throughput scaled well with the number of threads. For instance, with the smallest dataset 200k, the throughputs were 3858, 6901, 13130, 18103, and 23110 queries per second with 1, 2, 4, 8, and 16 threads, respectively. With the largest dataset 20mi, the throughputs were 215, 457, 896, 1605, and 2573 queries per second with 1, 2, 4, 8, and 16 threads respectively. This suggests that for up to 16 threads, the throughputs scale linearly with the number of threads. Results with the synthetic datasets are omitted.



(a) Twitter Datasets



(b) Synthetic Datasets



(c) Wikipedia Dataset

Figure 12: Total index building time: Pastri vs. IR-tree vs. I^3.

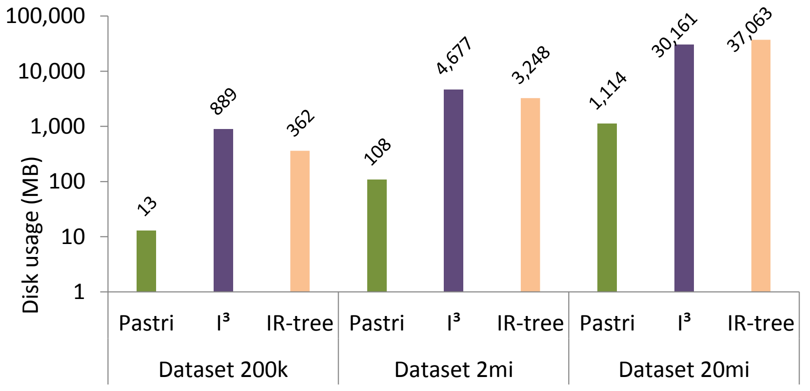


Figure 13: Disk usage (MB): Pastri vs. IR-tree vs. I^3—Twitter Datasets.

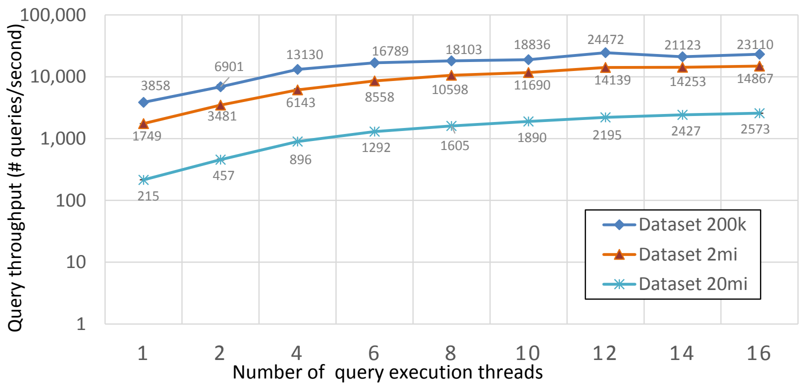
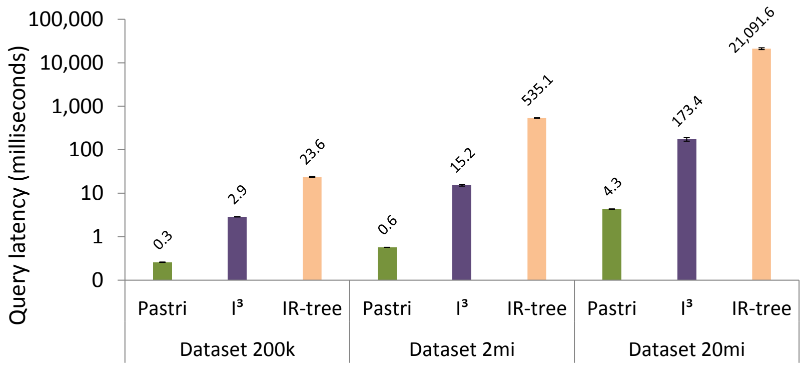


Figure 14: Query throughput (multi-threaded scalability)—Twitter Datasets.

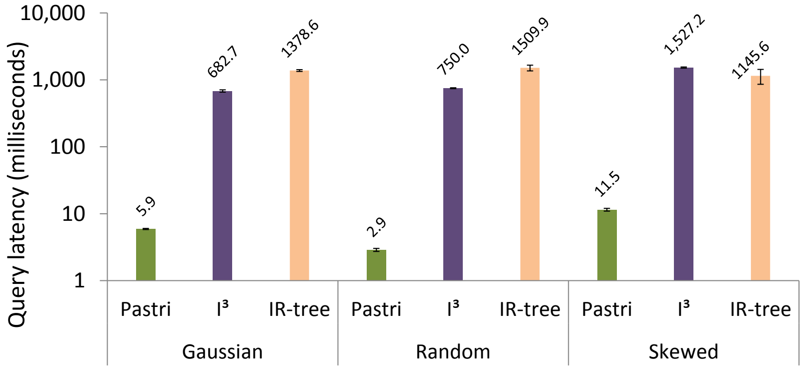
7.5 Query performance comparison

In this section, the goal is to analyze average query latency rather than query throughput. All reported query latencies in this section are the average of 1,000 queries as explained in section 7.1.2, with the remaining parameters having their default values except where noted. Unlike Pastri, IR-tree and I^3 indexes do not support multi-threaded query execution. Hence, we evaluate single threaded query performance of Pastri against these indexes. Note that unless otherwise specified, temporally relevant top-k spatial keyword query (TkSKQ) queries are evaluated. Hence, TkSKQ queries are used in the experiments in Sections 7.5 through Section 7.5.3. In Section 7.6, experiments are conducted to evaluate spatio-temporal textual top-k query (TkSTTQ) queries.

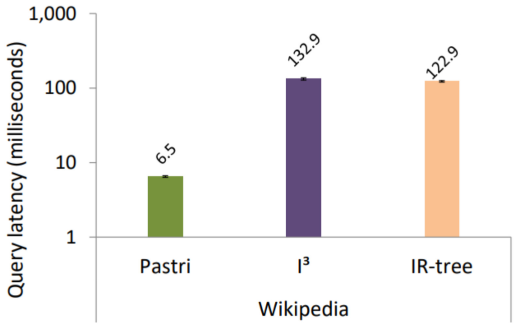
Figure 15(a) shows the TkSKQ query latency (in milliseconds) with Pastri, IR-tree, and I^3 index for the Twitter datasets. As can be observed in Figure 15(a), the average query latency is significantly less with Pastri than that with the other two. Specifically, the single threaded query latency with Pastri is one to two orders of magnitude lower than that of I^3 index, and at least two orders of magnitude lower than that of IR-tree. With 20mi, which is the largest dataset, the average query latency of Pastri is less than 5 milliseconds. With the 20mi dataset, the query latency for I^3 index is 173 milliseconds and for IR-tree it is 21



(a) Twitter Datasets



(b) Synthetic Datasets



(c) Wikipedia Dataset

Figure 15: Average query latency: Pastri (single threaded) vs. IR-tree vs. I^3.

seconds (or 21,092 milliseconds). An important factor behind Pastri's query performance is its in-memory index. Also, Pastri uses an in-memory cache of the recent document records, which allows it to avoid I/O costs significantly.

To analyze the query processing cost of IR-tree we profiled it with JVM Monitor and we show a breakdown of the time to execute queries in Figure 16. As Figure 16 shows,

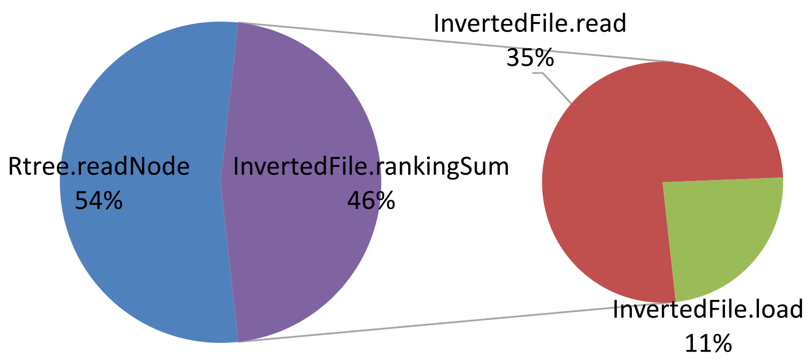


Figure 16: IR-tree query execution time breakdown.

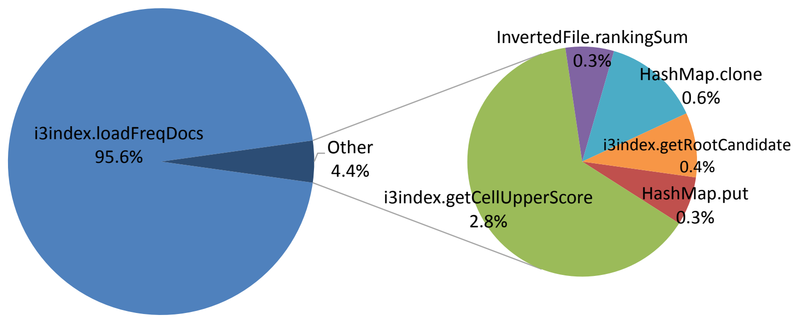


Figure 17: I^3 query execution time breakdown.

54% of the time is spent in *Rtree.readNode* function, which involves fetching the contents of Rtree nodes from disk. The remaining 46% is spent in processing the inverted file, of which 35% of the time is spent in the *InvertedFile.read* function. We do a similar profiling of query processing by the I^3 index and show the breakdown of execution time in Figure 17. With this, 95.6% of the time is spent in loading the index from disk.

The average query latencies comparing the indexes for the synthetic datasets are shown in Figure 15(b). Interestingly, for the dataset Skewed the query latency with I^3 is even worse than that with IR-tree. The text keywords are chosen from a Zipf distribution, so a few keywords occur very frequently in these synthetic datasets. Our analysis shows that I^3 spends about 70% of the query execution time in exploring the cells at different quad-tree levels in its attempt to do filtering, which leads to increased query latency.

For the Wikipedia dataset, the average query latency of TkSKQ queries with Pastri is compared against those with I^3 and IR-tree. As can be seen in Figure 15(c), Pastri is 20.4$\times$ faster than I^3 and 18.9$\times$ faster than IR-tree respectively. As in dataset Skewed, IR-tree performs better than I^3 in the Wikipedia dataset.

7.5.1 Vary selectivity

The selectivity of TkSKQ queries are varied in these experiments. The default selectivity used elsewhere is 20%. The higher the selectivity, the more processing cost that will be

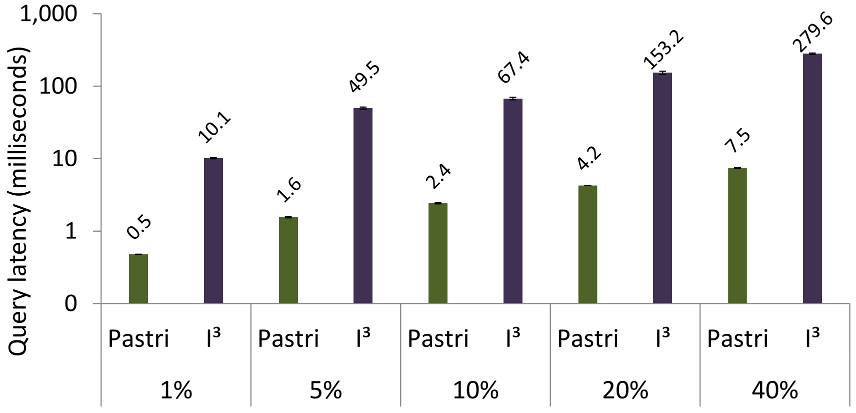


Figure 18: Average query latency (vary selectivity)—Twitter Datasets.

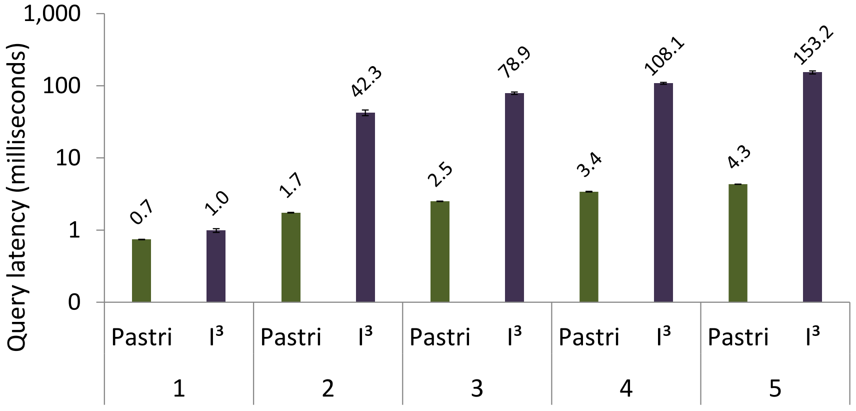


Figure 19: Average query latency (vary number of keywords)—Twitter Datasets.

incurred, as it increases the number of potential matching documents. In Figure 18 the average query latency is reported while varying 1%, 5%, 10%, 20%, and 40% selectivity. As can be seen, there is a clear trend showing an increase in query latency with the selectivity. However, with Pastri the magnitude of the overall increase is still quite small. Note that an increase in selectivity from 5% to 40% results in the average latency to rise from 1.6 milliseconds to 7.5 milliseconds only even with the largest dataset 20mi.

7.5.2 Vary number of query keywords

Next, we vary the number of query keywords and observe the average latency of TkSKQ queries. By default, in all other experiments the number of query keywords are set to 5. Figure 19 shows the average query latencies for the 3 datasets with number of query keywords varied 1, 2, 3, 4, and 5. As can be seen, the query latency decreases as the number of keywords are reduced. However, the magnitude of this different is only significant for the largest dataset 20mi.

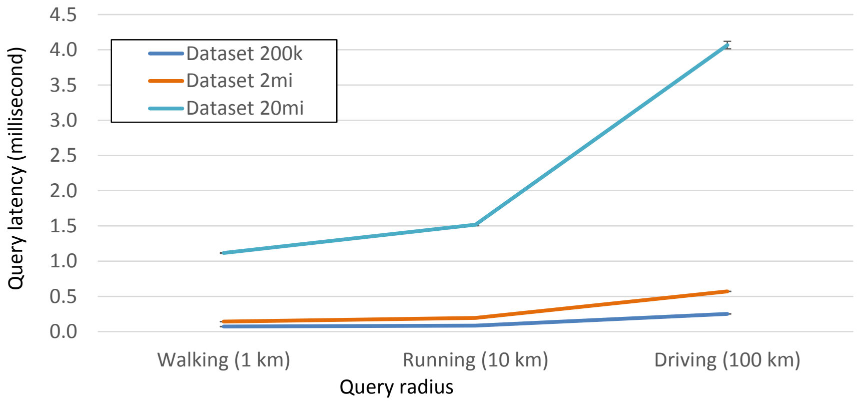


Figure 20: Average query latency (vary query radius).

7.5.3 Vary query radius

The default query radius used in all experiments involving TkSKQ query performance is 100 km. Next we vary the query radius as follows: 1 km, 10 km, and 100 km. These scenarios can be considered as use-cases for 3 different kinds of moving objects (as data sources): walking, biking, and driving, respectively. These parameters have natural correspondence to human travel queries (e.g. in Google maps). In Figure 20, we plot the average query latencies for these parameters with the Twitter datasets. As can be seen, the query latency decreases as the query radius gets smaller. However, even for the largest dataset 20mi and for maximum radius of 100 km, the latency is only 4.2 milliseconds. In fact, at this radius, the query latencies for the other two datasets (200k and 2mi) are below 1 millisecond. We believe that such latencies are quite acceptable for practical purposes. Moreover, in a real-world use case involving human travel, a radius longer than the Driving scenario may not occur, and even if it appears, the query latency will be small enough to be acceptable.

7.6 Query performance evaluation (TkSTTQ)

Here, we evaluate top-k spatio-temporal textual query (TkSTTQ) performance. In these experiments, we observe the average query latencies, as we vary the number of query keywords (as 1, 2, 3, 4, and 5) for each of the three Twitter datasets. As can be seen in Figure 21, the overall trend is that the query latency increases with the number of query keywords. However, even with 5 keywords, the query latencies are quite low (less than 1 millisecond), particularly for datasets 200k and 2mi. Even with the largest dataset 20mi, the query latency is 1.5 milliseconds with 1 keyword, and 9 milliseconds with 5 keywords.

8 Conclusion

Due to the rising volume of spatial-textual data, top-k spatial keyword search queries (TkSKQ) are growing in importance. There is a need to efficiently support these queries. Although several spatio-textual indexes support ad hoc (snapshot) TkSKQ, they are not able to operate on continuously generated data, as they need to build the index a priori. Re-

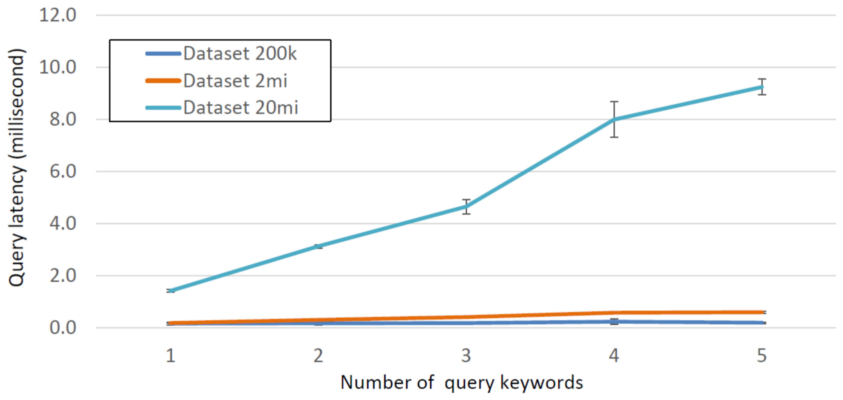


Figure 21: Average query latency (vary number of keywords) of TkSTTQ.

cently, a few systems have been proposed that can handle continuously generated document streams. However, these systems are not suitable to efficiently process queries while considering both document ranking based on textual similarity and spatial distance. Moreover, they do not consider the temporal relevance of the documents, even though we tend to value more recent objects, than older objects. Also, these indexes do not exploit parallelism well.

To address these issues, we introduced an integrated system for processing continuously generated document streams. Our system features a novel spatio-textual index, called Pastri. It utilizes a dynamic ranking scheme, DRTF-IDF, in order to retrieve documents that are the most temporally relevant based on a query criteria. Pastri supports multi-threaded processing of updates and query execution. We conducted extensive experimental evaluation with real-world datasets and synthetic datasets (that we created). Experimental results demonstrate that our system supports high update throughput with document streams and its ad hoc (snapshot) TkSKQ query latency is significantly lower than existing indexing approaches. We also show that the amortized cost to insert a data item into the our system is $\Theta(\frac{log_2 \frac{N}{BP}}{BP})$ I/Os, where N is the number of data items, P is the number of partitions in the pLSM store, and a block of data transferred to external memory holds B data items. Pastri also supports efficient top-k spatio-temporal textual (TkSTTQ) queries, demonstrating its applicability to a wider range of multi-dimensional search problems.

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), and by the University of New Brunswick Faculty of Computer Science

References

[1] Dictionary encoding. https://en.wikipedia.org/wiki/Dictionary_coder, 2020. Accessed: 2020-11-11.

[2] Internet live stats. www.internetlivestats.com, 2021. Accessed: 2021-02-16.

[3] AGGARWAL, A., AND VITTER, JEFFREY, S. The input/output complexity of sorting and related problems. *Communications of the ACM 31*, 9 (1988), 1116–1127. doi:10.1145/48529.48535.

[4] ALMASLUKH, A., AND MAGDY, A. Evaluating spatial-keyword queries on streaming data. In *Proceedings of the 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems* (2018), pp. 209–218. doi:10.1145/3274895.3274936.

[5] ALMASLUKH, A., AND MAGDY, A. Temporal geo-social personalized search over streaming data. In *Proceedings of the 27th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems* (2019), ACM, pp. 189–198. doi:10.1145/3347146.3359073.

[6] ARMEN, C. Bounds on the separation of two parallel disk models. In *Proceedings of the Fourth Workshop on I/O in Parallel and Distributed Systems: Part of the Federated Computing Research Conference* (New York, NY, USA, 1996), IOPADS '96, ACM, pp. 122–127. doi:10.1145/236017.236044.

[7] ARSENEAU, Y., GAUTAM, S., NICKERSON, B., AND RAY, S. STILT: Unifying spatial, temporal and textual search using a generalized multi-dimensional index. In *International Conference on Scientific and Statistical Database Management (SSDBM)* (2020). doi:10.1145/3400903.3400927.

[8] ASHAGRIE, M., TEKLI, J., TADDESSE, F. G., CHBEIR, R., AND TEKLI, G. Generic metadata representation framework for social-based event detection, description, and linkage. *Knowledge-Based Systems 188* (2019), 104817. doi:10.1016/j.knosys.2019.06.025.

[9] AsterixDB. https://github.com/apache/asterixdb, 2018.

[10] BENDER, M. A., FARACH-COLTON, M., JANNEN, W., JOHNSON, R., KUSZMAUL, B. C., PORTER, D. E., YUAN, J., AND ZHAN, Y. An introduction to B$^{\varepsilon}$-trees and write-optimization. *login; magazine 40*, 5 (2015).

[11] BUN, K. K., AND ISHIZUKA, M. Topic extraction from news archive using TF*PDF algorithm. In *Proceedings of the Third International Conference on Web Information Systems Engineering, 2002.* (2002), pp. 73–82. doi:10.1109/WISE.2002.1181645.

[12] BUSCH, M., GADE, K., LARSON, B., LOK, P., LUCKENBILL, S., AND LIN, J. J. Earlybird: Real-time search at Twitter. In *2012 IEEE 28th International Conference on Data Engineering* (2012), pp. 1360–1369. doi:10.1109/ICDE.2012.149.

[13] CHEN, L., CONG, G., CAO, X., AND TAN, K. Temporal spatial-keyword top-k publish/subscribe. In *2015 IEEE 31st International Conference on Data Engineering* (2015), pp. 255–266. doi:10.1109/ICDE.2015.7113289.

[14] CHEN, L., CONG, G., JENSEN, C. S., AND WU, D. Spatial keyword query processing: an experimental evaluation. *Proceedings of the VLDB Endowment 6*, 3 (2013), 217–228. doi:10.14778/2535569.2448955.

[15] CHENG, S., ARVANITIS, A., CHROBAK, M., AND HRISTIDIS, V. Multi-Query Diversification in Microblogging Posts. In *Proceedings of 17th International Conference on Extending Database Technology (EDBT)* (2014), pp. 133–144.

[16] CHOUDHURY, F. M., CULPEPPER, J. S., BAO, Z., AND SELLIS, T. Batch processing of top-k spatial-textual queries. *ACM Transactions on Spatial Algorithms and Systems (TSAS) 3*, 4 (2018), 1–40. doi:10.1145/3196155.

[17] CHRISTOFORAKI, M., HE, J., DIMOPOULOS, C., MARKOWETZ, A., AND SUEL, T. Text vs. Space: Efficient Geo-search Query Processing. In *Proceedings of the 20th ACM international conference on Information and knowledge management* (2011), pp. 423–432. doi:10.1145/2063576.2063641.

[18] CONG, G., JENSEN, C. S., AND WU, D. Efficient Retrieval of the Top-k Most Relevant Spatial Web Objects. *Proceedings of the VLDB Endowment 2*, 1 (2009), 337–348. doi:10.14778/1687627.1687666.

[19] http://www.dis.uniroma1.it/challenge9/, 2006.

[20] ERRA, U., SENATORE, S., MINNELLA, F., AND CAGGIANESE, G. Approximate TF–IDF based on topic extraction from massive message stream using the GPU. *Information Sciences 292* (2015), 143–161. doi:10.1016/j.ins.2014.08.062.

[21] FENG, W., ZHANG, C., ZHANG, W., HAN, J., WANG, J., AGGARWAL, C. C., AND HUANG, J. STREAMCUBE: hierarchical spatio-temporal hashtag clustering for event exploration over the twitter stream. In *2015 IEEE 31st international conference on data engineering* (2015), pp. 1561–1572. doi:10.1109/ICDE.2015.7113425.

[22] GROVER, R., AND CAREY, M. J. Data ingestion in AsterixDB. In *Proceedings of 18th International Conference on Extending Database Technology (EDBT)* (2015), pp. 605–616. doi:10.5441/002/edbt.2015.61.

[23] GUO, L., SHAO, J., AUNG, H. H., AND TAN, K.-L. Efficient continuous top-k spatial keyword queries on road networks. *GeoInformatica 19*, 1 (2014), 29–60. doi:10.1007/s10707-014-0204-8.

[24] HOANG-VU, T.-A., VO, H. T., AND FREIRE, J. A unified index for spatio-temporal keyword queries. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management* (2016), pp. 135–144. doi:10.1145/2983323.2983751.

[25] KHODAEI, A., SHAHABI, C., AND LI, C. Hybrid indexing and seamless ranking of spatial and textual features of web documents. In *International Conference on Database and Expert Systems Applications* (2010), pp. 450–466. doi:10.1007/978-3-642-15364-8_37.

[26] MAGDY, A., ALARABI, L., AL-HARTHI, S., MUSLEH, M., GHANEM, T. M., GHANI, S., AND MOKBEL, M. F. Taghreed: A system for querying, analyzing, and visualizing geotagged microblogs. In *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems* (2014), pp. 163–172. doi:10.1145/2666310.2666397.

[27] MAGDY, A., MOKBEL, M. F., ELNIKETY, S., NATH, S., AND HE, Y. Mercury: A memory-constrained spatio-temporal real-time search on microblogs. In *2014 IEEE 30th International Conference on Data Engineering* (2014), pp. 172–183. doi:10.1109/ICDE.2014.6816649.

[28] MAHMOOD, A., AND AREF, W. G. Query processing techniques for big spatial-keyword data. In *Proceedings of the 2017 ACM International Conference on Management of Data* (2017), SIGMOD '17, pp. 1777–1782. doi:10.1145/3035918.3054773.

[29] MAHMOOD, A. R., ALY, A. M., AND AREF, W. G. FAST: frequency-aware indexing for spatio-textual data streams. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018* (2018), pp. 305–316. doi:10.1109/ICDE.2018.00036.

[30] MAHMOOD, A. R., ALY, A. M., QADAH, T., REZIG, E. K., DAGHISTANI, A., MADKOUR, A., ABDELHAMID, A. S., HASSAN, M. S., AREF, W. G., AND BASALAMAH, S. Tornado: A distributed spatio-textual stream processing system. *Proceedings of the VLDB Endowment 8*, 12 (2015), 2020–2023. doi:10.14778/2824032.2824126.

[31] MAHMOOD, A. R., DAGHISTANI, A., ALY, A. M., TANG, M., BASALAMAH, S., PRABHAKAR, S., AND AREF, W. G. Adaptive processing of spatial-keyword data over a distributed streaming cluster. In *ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems* (2018), SIGSPATIAL '18, pp. 219–228. doi:10.1145/3274895.3274932.

[32] MARKOWETZ, A., YANG, Y., AND PAPADIAS, D. Keyword Search on Relational Data Streams. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data* (2007), pp. 605–616. doi:10.1145/1247480.1247548.

[33] http://norvig.com/ngrams/count_1w100k.txt, 2011.

[34] NEGI, D., RAY, S., AND LU, R. Pystin: Enabling secure LBS in smart cities with privacy-preserving top-k spatial-textual query. *IEEE Internet of Things Journal 6*, 5 (2019), 7788–7799. doi:10.1109/JIOT.2019.2902483.

[35] O'NEIL, PATRICK AND CHENG, EDWARD AND GAWLICK, DIETER AND O'NEIL, ELIZABETH. The Log-structured Merge-tree (LSM-tree). *Acta Informatica 33*, 4 (1996), 351–385. doi:10.1007/s002360050048.

[36] PONTE, J. M., AND CROFT, W. B. A Language Modeling Approach to Information Retrieval. In *SIGIR* (1998), pp. 275–281.

[37] RAY, S., AND NICKERSON, B. G. Dynamically ranked top-k spatial keyword search. In *Proceedings of the Third International ACM SIGMOD Workshop on Managing and Mining Enriched Geo-Spatial Data* (2016), pp. 6:1–6:6. doi:10.1145/2948649.2948655.

[38] RAY, S., AND NICKERSON, B. G. Improving parallel performance of temporally relevant top-k spatial keyword search. In *ACM SIGSPATIAL Workshop on Recommendations for Location-based Services and Social Networks* (2018), LocalRec, pp. 5:1–5:4. doi:10.1145/3282825.3282830.

[39] Reed, J. W., Elmore, M. T., Potok, T. E., Hurson, A. R., Jiao, Y., and Klump, B. A. TF-ICF: A new term weighting scheme for clustering dynamic data streams. In *2006 5th International Conference on Machine Learning and Applications (ICMLA'06)* (2006), pp. 258–263. doi:10.1109/ICMLA.2006.50.

[40] Rocha-Junior, J. a. B., Gkorgkas, O., Jonassen, S., and Nørvåg, K. Efficient Processing of Top-k Spatial Keyword Queries. In *International Symposium on Spatial and Temporal Databases* (2011), pp. 205–222. doi:10.1007/978-3-642-22922-0_13.

[41] Salton, G., and Buckley, C. Term-weighting approaches in automatic text retrieval. *Information processing & management 24*, 5 (1988), 513–523. doi:10.1016/0306-4573(88)90021-0.

[42] Salton, G., Wong, A., and Yang, C. S. A Vector Space Model for Automatic Indexing. *Communications of the ACM 18*, 11 (1975), 613–620. doi:10.1145/361219.361220.

[43] Scully, J. Serving over 1 billion documents per day with docstore v2. https://engineering.indeedblog.com/blog/2013/10/serving-over-1-billion-documents-per-day-with-docstore-v2/, 1988.

[44] Skovsgaard, A., Sidlauskas, D., and Jensen, C. S. Scalable top-k spatio-temporal term querying. In *ICDE* (2014), pp. 148–159.

[45] Sohail, A., Cheema, M. A., and Taniar, D. Geo-social temporal top-k queries in location-based social networks. In *Databases Theory and Applications. ADC 2020* (2020), Springer International Publishing, pp. 147–160.

[46] Tekli, J., Chbeir, R., Traina, A. J. M., and Traina, C. SemIndex+: A semantic indexing scheme for structured, unstructured, and partly structured data. *Knowledge-Based Systems 164* (2019), 378–403. doi:10.1016/j.knosys.2018.11.010.

[47] Theodoridis, Y., Silva, J. R. O., and Nascimento, M. A. On the generation of spatiotemporal datasets. In *International Symposium on Spatial Databases* (1999), pp. 147–164. doi:10.1007/3-540-48482-5_11.

[48] Vaid, S., Jones, C. B., Joho, H., and Sanderson, M. Spatio-textual indexing for geographical search on the web. In *International Symposium on Spatial and Temporal Databases* (2005), pp. 218–235. doi:10.1007/11535331_13.

[49] Vitter, J. S. Algorithms and data structures for external memory. *Foundations and Trends® in Theoretical Computer Science 2*, 4 (2008), 305–474. doi:10.1561/0400000014.

[50] Vitter, J. S., and Shriver, E. A. Algorithms for parallel memory, I: Two-level memories. *Algorithmica 12*, 2-3 (1994), 110–147. doi:10.1007/BF01185207.

[51] Wang, X., Zhang, Y., Zhang, W., Lin, X., and Wang, W. AP-Tree: efficiently support location-aware publish/subscribe. *The VLDB Journal 24*, 6 (2015), 823–848. doi:10.1007/s00778-015-0403-4.

[52] https://dumps.wikimedia.org/enwiki/latest/, 2018.

[53] YAN, H., DING, S., AND SUEL, T. Inverted Index Compression and Query Processing with Optimized Document Ordering. In *Proceedings of the 18th international conference on World wide web* (2009), pp. 401–410. doi:10.1145/1526709.1526764.

[54] ZHANG, D., TAN, K.-L., AND TUNG, A. K. H. Scalable Top-k Spatial Keyword Search. In *Proceedings of the 16th international conference on extending database technology* (2013), pp. 359–370. doi:10.1145/2452376.2452419.