Improving Parallel Performance of Temporally Relevant Top-K Spatial Keyword Search

Suprio Ray and Bradford G. Nickerson
University of New Brunswick, Fredericton, Canada

ABSTRACT

With the rapid growth of geo-tagged documents, top-k spatial keyword search queries (TkSKQ) have attracted a lot of attention and a number of spatio-textual indexes have been proposed. While some indexes support real-time updates over continuously generated documents, they do not support queries that simultaneously consider temporal relevance, textual similarity ranking and spatial location. Existing indexes also have limited capability to exploit parallelism.

To address these issues, we introduce a novel parallel index, called Pastri (PArallel Spatio-Textual adaptive Ranking-based Index), which can be incrementally updated based on live spatio-textual document streams. Pastri uses a dynamic ranking scheme to retrieve the top-k objects that are most temporally relevant at the time of a query execution. We have built a system in which we integrate Pastri along with a persistent document store and several thread pools to exploit parallelism at various levels. Experimental evaluation demonstrates that our system can support high document update throughput and low latency with TkSKQ queries.

CCS CONCEPTS

• **Information systems → Top-k retrieval in databases**; **Retrieval efficiency**; **Search engine indexing**;

KEYWORDS

spatio-textual; index; temporal relevance; spatial keyword search

ACM Reference Format:
Suprio Ray and Bradford G. Nickerson. 2018. Improving Parallel Performance of Temporally Relevant Top-K Spatial Keyword Search. In *2nd ACM SIGSPATIAL Workshop on Recommendations for Location-based Services and Social Networks (LocalRec'18), November 6, 2018,Seattle, WA, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3282825.3282830

1 INTRODUCTION

The popularity of Location-Based Services (LBS) and social media use is contributing to deluge of spatio-textual data. Often, the value of such data depends on the "recency" or "temporal relevance" of the generated documents. For instance, during a natural disaster, emergency response teams can act quickly on recently reported incidents. The high volume and velocity of generated spatio-textual documents requires novel approaches to indexing and query processing. Spatio-textual queries can be classified [10] into the following categories: filter, top-k, collective, and other queries. A top-k spatial keyword query (TkSKQ) retrieves k objects based on both the distance of the generated documents (objects) from the query location and textual relevance.

Due to the popularity of geo-textual search, a number of spatio-textual indexes have been proposed. In a comprehensive study of 12 of these indexes, Chen et al. [3] report that a few of the indexes such as [5, 15, 18] support TkSKQ. However, these spatio-textual indexes do not support real-time data ingestion. Indexing systems such as I^3 [18] and IR-tree [5] are tree data structure based. Such organization offers limited parallelism, and are generally not scalable with update-heavy workloads. In recent times, a number of indexing approaches have been proposed that focused on efficient processing of textual streams. Earlybird [1], focused on real-time text search, but does not support spatial queries. Taghreed [8] can execute arbitrary queries on microblog streams. It supports a few types of queries such as spatio-temporal boolean range keyword search, and top-k frequent keyword query. Taghreed does not support ranked TkSKQ or a ranking scheme that considers both spatial and textual relevance. Mercury [9], proposed an in-memory index for real-time support of top-k spatio-temporal queries on microblog data streams. Mercury's ranking scheme, however, only considers spatial and temporal relevance, not textual relevance. The goals that we share with systems such as Earlybird and Taghreed, are high throughput ingestion of spatio-textual streams and ad hoc (arbitrary) queries with interactive response time. Unlike these systems, however, our approach supports ad hoc TkSKQ queries and a ranking scheme that considers spatial and textual relevance, and temporal recency.

Note that a number of location-aware publish-subscribe systems have been proposed that support continuous spatio-temporal queries over data streams. A continuous query is different from an ad hoc query, as a continuous query needs to be registered first, and then the answer to that query is evaluated over time on the the incoming data stream. Some of the indexing approaches to support continuous queries over spatio-textual data streams include TaSK [2], AP-Tree [17] and Tornado [11]. These approaches are *not relevant* to our system, as we support *ad hoc* TkSKQ queries on textual data streams.

To support real-time ingestion of document streams and efficient TkSKQ over such data, we introduce a parallel index called PASTRI (PArallel Spatio-Textual adaptive Ranking-based Index) or "Pastri". This work extends our previous work [14], in which we introduced an approximate temporally relevant dynamic ranking scheme for real-time document streams. Pastri is designed to be highly scalable, and support parallel document updates and concurrent query execution. To support high throughput disk-based storage, we present

LocalRec'18, November 6, 2018, Seattle, WA, USA

ACM ISBN 978-1-4503-6040-1/18/11…$15.00
https://doi.org/10.1145/3282825.3282830

a simple yet effective extension to the log-structured merge-tree (or LSM-tree) [12] that we call pLSM (partitioned LSM-tree) store. We present three load-balancing algorithms to handle data skew. Our index can be considered as a hybrid between tree and grid indexing approaches. The updates are handled by a component that combines the in-memory grid with inverted lists, along with an in-memory table which is backed by an in-memory cache and on-disk storage. Queries are directed to the in-memory tree component that can perform efficient circular range search from the query location. Pastri uses our dynamic ranking measure to calculate a 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 search returns the most recent relevant objects.

We implemented a prototype of our system and evaluated it with a real world Twitter dataset. Our approach delivers high performance for a single server class machine with large main memory and multiple cores, since servers with hundreds of GB or a few TB are common today. We present a thorough evaluation and show (see Section 4) that Pastri can ingest and index about 200,000 documents/second. In contrast, Taghreed [8] reportedly can ingest up to 32,000 microblogs/second; [8] does not report any query performance. Mercury [9] can ingest 64,000 microblogs/second. Query latencies achieved by Pastri are comparable to those of Mercury or lower. Furthermore, when compared with two popular spatio-textual indexes, IR-tree and I^3, Pastri's TkSKQ query performance is significantly better.

2 RELATED WORK

First, we describe research on spatio-textual indexing approaches and then mention previous work on indexing document streams.

2.1 Spatio-textual Index

Depending on the structure of spatio-textual indexes, they can be classified into three groups: R-tree based indexes, grid-based indexes and space filling curve based indexes.

The IR-tree [5] is one of the R-tree based approaches that support TkSKQ. It integrates an R-tree with an inverted file. The S2I [15] index uses two different approaches for frequent terms and infrequent terms. For infrequent keywords, all the elements in the inverted file are stored sequentially for efficient I/O. Otherwise, an aggregated R-tree (aR-tree) is used for pruning. The I^3 index [18] uses a similar textual partitioning approach as S2I. However, they use a Quadtree instead of an R-tree for spatial indexing.

SFC-QUAD [4] is one among only a few non-R-tree based approaches that support TkSKQ. In each inverted list the document ids are ordered based on their position in a Z-curve. The grid-based indexes incorporate a grid index with a text index. Among these, the two most prominent ones are ST & TS [16] and SKIF [7]. The grid-based approaches can only support boolean range queries, but not TkSKQ, as noted by [3].

2.2 Indexing spatio-textual data streams

Recent work on real-time search over a massive stream focusses on the requirement for immediate search with very low latency updates. The Earlybird [1] system focusses on real-time search of text, and does not support spatial queries. StreamCube [6] supports real-time top-k spatio-temporal ranking of hashtags in the Twitter stream. Unlike Pastri, however, StreamCube does not support user-defined top-k spatial keyword search.

Taghreed [8] supports several types of queries (e.g. find the top-k most frequent keywords used within given spatial and temporal ranges), but it does not support ranked top-k spatial keyword queries. Mercury [9] supports top-k spatio-temporal queries, but its ranking scheme is based on spatial and temporal relevance, which does not consider textual relevance.

Our focus is on supporting ad hoc ranked top-k spatial keyword queries. Therefore, publish-subscribe approaches [2, 11, 17] that support continuous spatio-temporal queries over data streams are not relevant to this discussion. Also, our prior work [13], is not relevant in this context, as it did not support spatio-textual data. Present work extends our previous research [14].

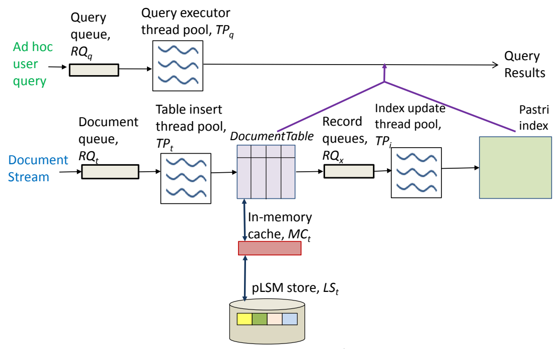

Figure 1: System architecture

3 PASTRI SPATIO-TEXTUAL INDEX

In this section we describe our system organization, internal data structures and the update and query processing algorithms.

3.1 System organization

Our system has a highly multi-threaded organization. Figure 1 shows the update and query processing workflows. When a new geo-tagged document, such as a tweet, $o = (o.id, o.l, o.d, t_o)$ is received from the document stream by our system, it is enqueued into RQ_t. The document is retrieved from the queue by one of the table insert threads in the thread pool TP_t. It creates a document record which is inserted into the table *DocumentTable*. Upon insertion, a unique record id, *RID*, is generated by incrementing a variable with an atomic operation.

The table is backed by a partitioned LSM-tree store (described in Section 3.3) on disk. Moreover, there is also an in-memory cache to store most recent documents. So, an insert operation on the *DocumentTable*, causes a document record to be inserted into the in-memory cache and then onto the disk. Next an entry is created with the *RID* which is enqueued in one of the queues in RQ_x. An index update thread from thread pool TP_i dequeues the record and processes it. The index update process is described in section 3.4.

Our system can handle many concurrent queries. When a new query is submitted into the system, it is enqueued in RQ_q. Then it is picked up by one of the threads in the thread pool TP_q and processed by it. The query processing is outlined in section 3.5.

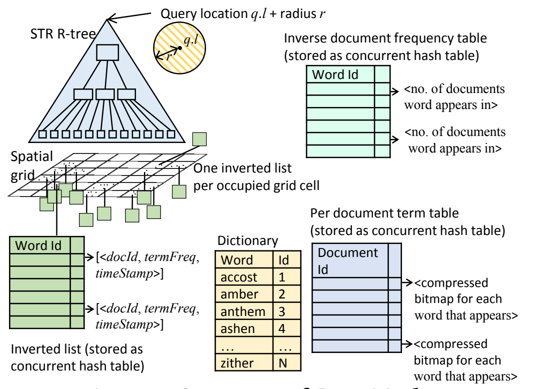
Query location $q.l$ + radius r
STR R-tree
$q.l$
r
Inverse document frequency table
(stored as concurrent hash table)
Word Id
<no. of documents word appears in>
<no. of documents word appears in>
Spatial grid
One inverted list per occupied grid cell
Per document term table
(stored as concurrent hash table)
Word Id
[<docId, termFreq, timeStamp>]
[<docId, termFreq, timeStamp>]
Inverted list (stored as concurrent hash table)
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>

Figure 2: Structure of Pastri index

3.2 Index structure

As shown in Figure 2, our index, Pastri, can be considered as a hybrid between tree and grid indexing approaches. It organizes the spatial domain as a grid (*GRID*) by dividing it into regular sized grid cells. The cells of *GRID* are indexed by an STR R-tree.

For each cell an inverted list (*ILIST*) is maintained for the documents that are indexed at that cell. Whenever a new document object is received, the corresponding cell is determined from its location (i.e. $o.l.lat$, $o.l.lon$). The *ILIST* is a concurrent hash table, in which the key is a word id and value is a list of tuples. Each word id corresponds to the numeric identifier in the global word dictionary (*DICT*). Each tuple consists of 3 entries: the document id, the term frequency - the number of times a word appears in that document, and the timestamp when the document was generated.

The dictionary, *DICT*, is a global data structure. In addition, there are three global data structures: an inverse document frequency table (*IDFT*), a per document term table (*PDTT*) and a global document counter (*DC*). Any time a new document object is indexed, the counter *DC* is incremented. The *IDFT* keeps track of inverse document frequency for each word i.e. the number of unique documents in which a word appears. The *PDTT* data structure is used to maintain a concise representation of each document. It is a concurrent hash table in which the key is a document id and value is a compressed bitmap (*CBMAP*). A position in the bitmap, *p*, is set if a word appears in the document whose document id in *DICT* is *p*. *CBMAP* is compressed to be memory efficient.

3.3 Storage organization

Since the *DocumentTable* is backed by a persistent disk-resident storage, a key consideration to improve the parallel performance is how to minimize disk I/Os. The log-structured merge-tree (or LSM-tree) [12] is an insert efficient data structure that provides indexed access to files in the form of key-value pairs. A known drawback of the LSM-tree is that it does not scale well with multiple CPU cores.

To address the scalability issue of the LSM-tree, we introduce a simple but effective extension that we call partitioned LSM-tree store or pLSM store (shown in Figure 1). The pLSM store consists of multiple LSM-trees $\{P_i | 1 \leq i \leq p\}$, each with a dedicated queue and an insert thread. When a new insert requests comes for an object $o \in O$, the pLSM store internally maps o into a partition i and inserts it into the corresponding queue. The dedicated insert thread then inserts o into partition i. In section 4.2.2, we show that insert throughput with pLSM store scales well.

Table 1: Twitter datasets

Dataset name	Num. of tuples	Average num. of keywords	Max num. 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

3.4 Index update processing

To update the index with a new document object, first its location field $o.l$ is used to determine which grid cell it belongs to. Then the corresponding inverted list object, *ILIST*, is updated by registering a new entry for each word in that document. The global data structures *IDFT*, *PDTT* and *DC* are updated.

3.4.1 Load-balancing and handling skew. The objective of load-balancing is to minimize the variation in the number of geo-tagged documents processed by the update processing threads. We explore 3 load-balancing algorithms: Round-robin, Work-stealing, and Affinity. The Round-robin load-balancing approach inserts the next document record into a queue in RQ_x in a round-robin fashion. Then any index insert thread from the thread pool TP_i just picks up the next record from its associated queue. In the Work-stealing approach, threads in TP_i are not associated to any queue in RQ_x; rather, each thread actively inspects every queue in RQ_x for an available record to process. With the Affinity approach, a mapping of grid cells to index insert threads, *Gcell2ThreadMapping*, is created by an offline algorithm *AffinityAssignment* (skipped due to space constraint), such that the standard deviation of the object density of each thread is less than a threshold. *AffinityAssignment* performs a density sampling of each cell periodically 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*.

3.5 Query processing

For a TkSKQ query q, the index is used to perform a circular range search in which the point of origin is $q.l$ and the initial search radius r. For each cell returned by the index, the corresponding inverted list *ILIST* is consulted to compute the ranking. This involves first computing the textual relevance score for each document that has a match for one of the query keywords. Then, for each of those documents, the spatial relevance score and the combined score is calculated. A priority queue *PQueue* is used to maintain the current top-k documents. If the combined score of a document is less than the score of the k-th document in *PQueue*, it is enqueued.

4 EVALUATION

In this section we evaluate our system in various settings.

4.1 Experimental setup

We conducted our performance evaluation with three Twitter datasets. The details are presented in Table 1. The experiments were conducted on a machine with 128 GB memory, 16 AMD Opteron processing cores, running Red Hat Enterprise Linux 4.8 64-bit.

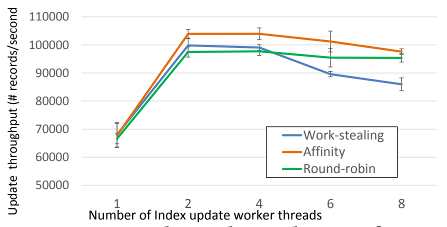

Figure 3: Update throughput of Pastri with different load-balancing algorithms (dataset 20mi)

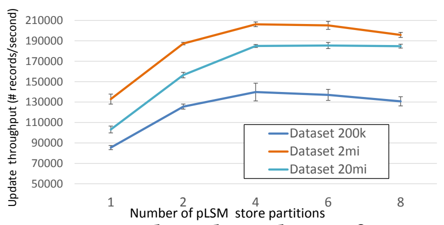

Figure 4: Update throughput of Pastri with different number of storage partitions

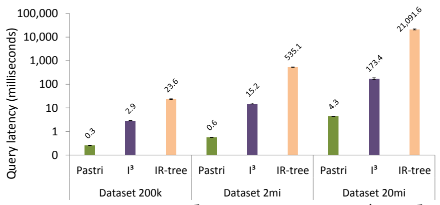

Figure 5: Query latency: Pastri (single threaded) vs. IR-tree vs. I^3

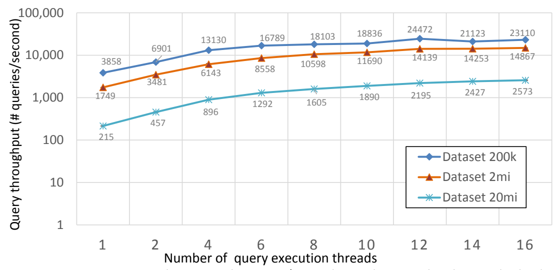

Figure 6: Query throughput (multi-threaded scalability)

4.2 Update Performance

4.2.1 Load-balancing algorithm. We evaluate the impact of the three load-balancing algorithms (Work-stealing, Affinity and Round-robin) on the update throughput of Pastri. To isolate different contributing factors, we fixed the number of partitions in pLSM store to 1. In Figure 3, we plot the observed update throughput with Twitter 20mi dataset as we vary the number of index update threads. As can be seen, Affinity achieves the best throughput in all cases. We also conducted experiments (omitted) by varying the number of grid cells to 64, 256, 1024 and 4096 cells. We found that the throughput remains relatively stable regardless of the grid resolution.

4.2.2 pLSM store partitions. We vary the number of partitions in the pLSM store, while fixing the grid resolution to 1024 and the load-balancing algorithm to Affinity. Figure 4 shows the update throughputs with the three Twitter datasets. As can be seen, Pastri achieves high update throughput. For instance, the throughput is about 200,000 documents/second with 2mi and 20mi datasets.

4.3 Query performance

4.3.1 Query throughput (parallel performance). We executed 1 million TkSKQ queries with Pastri while varying the number of query execution threads 1, 2, 4, 6, 8, 10, 12, 14 and 16. Figure 6 (in *log scale*) demonstrates that the query throughputs scale well with the number of threads. For instance, the throughput is 2573 queries per second with 16 threads for dataset 20mi.

4.3.2 Query latency. Since IR-tree and I^3 index do not support parallel query execution, we only compare single threaded query performance of Pastri against them. The average latencies of 1000 TkSKQ queries with Pastri, IR-tree and I^3 index for the Twitter datasets are shown in Figure 5. To plot these charts, each of the 1000 queries were executed with these parameters: query selectivity 20%, number of keywords 5 and query radius 100 km. As can be seen, Pastri significantly outperforms the other two. Pastri's single threaded query latency is one to two orders of magnitude faster than that of I^3 index, and at least two orders of magnitude faster than that of IR-tree. Even with the largest dataset 20mi, the average latency with Pastri is less than 5 milliseconds.

5 CONCLUSION

We have presented a system for processing continuously generated document streams. It incorporates a novel spatio-textual index, Pastri, that retrieves most temporally relevant documents using a dynamic ranking scheme. Pastri is a parallel spatio-textual index that exploits concurrency to accelerate update and query performance. With extensive experimental evaluation we demonstrate that our system offers superior performance over existing indexing approaches supporting top-k spatial keyword search queries.

REFERENCES

[1] Michael Busch, Krishna Gade, Brian Larson, Patrick Lok, Samuel Luckenbill, and Jimmy J. Lin. 2012. Earlybird: Real-Time Search at Twitter. In *ICDE*. 1360–1369.
[2] Lisi Chen, Gao Cong, Xin Cao, and Kian-Lee Tan. 2015. Temporal Spatial-Keyword Top-k publish/subscribe. In *ICDE*. 255–266.
[3] Lisi Chen, Gao Cong, Christian S. Jensen, and Dingming Wu. 2013. Spatial keyword query processing: an experimental evaluation. In *PVLDB*. 12.
[4] Maria Christoforaki, Jinru He, C. Dimopoulos, Alexander Markowetz, and Torsten Suel. 2011. Text vs. Space: Efficient Geo-search Query Processing. In *CIKM*.
[5] Gao Cong, Christian S. Jensen, and Dingming Wu. 2009. Efficient Retrieval of the Top-k Most Relevant Spatial Web Objects. *VLDB* 2, 1 (2009).
[6] Wei Feng, Chao Zhang, Wei Zhang, Jiawei Han, Jianyong Wang, Charu C. Aggarwal, and Jianbin Huang. 2015. STREAMCUBE: Hierarchical spatio-temporal hashtag clustering for event exploration over the Twitter stream. In *ICDE*.
[7] Ali Khodaei, Cyrus Shahabi, and Chen Li. 2010. Hybrid Indexing and Seamless Ranking of Spatial and Textual Features of Web Documents. In *DEXA*. 450–466.
[8] Amr Magdy, Louai Alarabi, Saif Al-Harthi, Mashaal Musleh, Thanaa M. Ghanem, Sohaib Ghani, and Mohamed F. Mokbel. 2014. Taghreed: A System for Querying, Analyzing, and Visualizing Geotagged Microblogs. In *SIGSPATIAL GIS*.
[9] Amr Magdy, Mohamed F. Mokbel, Sameh Elnikety, Suman Nath, and Yuxiong He. 2014. Mercury: A memory-constrained spatio-temporal real-time search on microblogs. In *ICDE*. 172–183.
[10] Ahmed Mahmood and Walid G. Aref. 2017. Query Processing Techniques for Big Spatial-Keyword Data. In *SIGMOD*.
[11] A. R. Mahmood, A. M. Aly, T. Qadah, E. K. Rezig, A. Daghistani, A. Madkour, A. S. Abdelhamid, M. S. Hassan, W. G. Aref, and S. Basalamah. 2015. Tornado: A Distributed Spatio-textual Stream Processing System. *Proc. VLDB Endow.* (2015).
[12] O'Neil, Patrick and Cheng, Edward and Gawlick, Dieter and O'Neil, Elizabeth. 1996. The Log-structured Merge-tree (LSM-tree). *Acta Inf.* (1996).
[13] Suprio Ray, Rolando Blanco, and Anil K. Goel. 2014. Supporting Location-Based Services in a Main-Memory Database. In *MDM*.
[14] Suprio Ray and Bradford G. Nickerson. 2016. Dynamically ranked top-k spatial keyword search. In *GeoRich@SIGMOD*. 6:1–6:6.
[15] João B. Rocha-Junior, Orestis Gkorgkas, Simon Jonassen, and Kjetil Nørvåg. 2011. Efficient Processing of Top-k Spatial Keyword Queries. In *SSTD*. 18.
[16] Subodh Vaid, Christopher B. Jones, Hideo Joho, and Mark Sanderson. 2005. Spatio-textual Indexing for Geographical Search on the Web. In *SSTD*.
[17] Xiang Wang, Ying Zhang, Wenjie Zhang, Xuemin Lin, and Wei Wang. 2015. AP-Tree: efficiently support location-aware Publish/Subscribe. *VLDB J.* (2015).
[18] Dongxiang Zhang, Kian-Lee Tan, and Anthony K. H. Tung. 2013. Scalable Top-k Spatial Keyword Search. In *EDBT*. 359–370.