Toward Efficient Navigation of Massive-Scale Geo-Textual Streams

Chengcheng Yang, **Lisi Chen**[*], **Shuo Shang**[*], **Fan Zhu**, **Li Liu** and **Ling Shao**
Inception Institute of Artificial Intelligence
{chengcheng.yang, lisi.chen, shuo.shang, fan.zhu, li.liu, ling.shao}@inceptioniai.org

Abstract

With the popularization of portable devices, numerous applications continuously produce huge streams of geo-tagged textual data, thus posing challenges to index geo-textual streaming data efficiently, which is an important task in both data management and AI applications, e.g., real-time data streams mining and targeted advertising. This, however, is not possible with the state-of-the-art indexing methods as they focus on search optimizations of static datasets, and have high index maintenance cost. In this paper, we present NQ-tree, which combines new structure designs and self-tuning methods to navigate between update and search efficiency. Our contributions include: (1) the design of multiple stores each with a different emphasis on write-friendness and read-friendness; (2) utilizing data compression techniques to reduce the I/O cost; (3) exploiting both spatial and keyword information to improve the pruning efficiency; (4) proposing an analytical cost model, and using an online self-tuning method to achieve efficient accesses to different workloads. Experiments on two real-world datasets show that NQ-tree outperforms two well designed baselines by up to $10\times$.

1 Introduction

With the prevalence of smartphones, massive amounts of geo-textual data are being generated at a rapid pace. Geo-textual data has become a critical building block for today's AI researches and applications, especially in the field of real-time data streams analysis. For instance, many data mining tasks, e.g., geographical topic discovery [Hong *et al.*, 2012], local event detection [Walther and Kaisser, 2013; Chen and Shang, 2019], POI annotation with geo-tagged Tweets [Zhao *et al.*, 2016; Chen *et al.*, 2014], and location-based recommendation [Yin *et al.*, 2013; Liu *et al.*, 2013], have been conducted on the data from location based social services.

[*]Lisi Chen and Shuo Shang are corresponding authors.

Another example is targeted advertising. Advertisers register millions of ads, and highly targeted ads are pushed to users in real time according to their interests and positions. For example, if a user is interested in *Chinese spicy* food, based on the user's current location derived from his/her smartphone, the server would push restaurant ads within a specific range to the user. In this scenario, new restaurant ads are registered continuously, and users can change their interests (keywords) and positions, which results in different search regions. For example, if a user is interested in *Chinese spicy* food, the search region may be 5km. However, if the user is interested in *wedding photography*, the search region might be 20km. These requirements raise the question of how to offer efficient support for both data updates and searches on the changing workloads.

Many recent studies have addressed the problem of designing search efficient indexes. Existing studies can be broadly classified into *spatial-first* indexes and *keyword-first* indexes [Chen *et al.*, 2013]. At their core, these approaches employ sophisticated data structures or data partitioning schemes to improve the pruning efficiency.

Most of *spatial-first* indexes use the R-tree as the basic index structure, and embed inverted indexes into each node [Hariharan *et al.*, 2007; Cong *et al.*, 2009; Li *et al.*, 2011]. However, they are inefficient for streaming data since each update leads to propagated updates of inverted indexes. The *keyword-first* indexes usually organize objects by spatial indexes for each keyword. For example, IL-Quadtree [Zhang *et al.*, 2016] builds a Quadtree for each keyword and utilizes costly data partitions to improve the pruning efficiency. However, some key parameters, such as the leaf size, minimal Quadtree depth, should be carefully chosen according to the search logs and datasets, which makes it not practical for streaming data. SFC-QUAD [Christoforaki *et al.*, 2011] combines the space filling curve and data compressions to reduce the scanning cost of inverted lists. Unfortunately, it only applies to static datasets.

In this paper, we present NQ-tree (Navigable Quadtree), which is tailored to optimize the update performance while preserving the search efficiency. Specifically, we propose to use the Quadtree with low maintenance cost for space decomposition. Then we adopt the *cascading update* technique [Arge, 2003] for efficient updates. In an NQ-tree, each node is attached with a log store to cache updates, and then all updates are propagated to leaf nodes in a cascading manner, so that the write cost is amortized. Each leaf has a data store to accommodate streaming data, and is dynamically merged with the log store according to the underlying workloads. To support efficient searches over multiple log stores, we design space-efficient auxiliary structures that combine spatial and keyword information to prune the log pages that do not meet the search constraints. Further, we use Bloom filters that contain summary keyword information for data store pruning. To navigate between update and search efficiency in face of varying workloads, we analytically estimate the cost of index adjustments, and use an online algorithm to dynamically empty log stores and split/merge leaf nodes adaptively. In addition, we utilize data compression techniques to further reduce the I/O cost. In summary, we make the following contributions:

- We develop a novel index with multiple stores, each with different optimization purposes such as high update performance or inexpensive search cost.
- We design space-efficient auxiliary structures that combine spatial and keyword information to enhance the pruning capacity.
- We propose a cost based self-tuning method to adapt the index to workload shifts.
- We conduct experiments on two real-world datasets. The experimental results show that NQ-tree outperforms two baselines by up to an order of magnitude.

2 Preliminaries

2.1 Problem Definition

The geo-textual object stream $\mathcal{O} = \{o_1, \cdots, o_n\}$ is defined as a sequence of objects $o_i = \{ID_i, \psi_i, l_i\}$ which arrive continuously. Each object o_i is identified by a unique ID_i, and is represented as a textual message ψ_i with geo-location l_i. A geo-textual search $q = \langle r, \psi \rangle$ has two components, where $q.r$ represents a spatial region and $q.\psi$ represents a set of keywords. The answer to q is a list of geo-textual objects that are in the search region $q.r$ and whose descriptions contain the set of search keywords $q.\psi$. We tackle the problem of processing a stream of geo-textual objects continuously, and users can issue geo-textual searches to get all answers in real time. The number of streaming objects can be very large, and the number of search answers may vary with data distributions.

2.2 Related Work

Geo-textual search. Hariharan et al. [Hariharan *et al.*, 2007], Cong et al. [Cong *et al.*, 2009] and Li et al. [Li *et al.*, 2011] augment the nodes of an R-tree with keywords summarization from subtrees, which is used for efficient nodes pruning. Christoforaki et al. [Christoforaki *et al.*, 2011] combines inverted file with the space filling curve. Zhang et al. [Zhang *et al.*, 2016] builds a partition based Quadtree for each keyword. Besides applying spatial and keywords constraints, some variants focus on ranking objects based on a scoring function that considers both textual relevance (similarity) and spatial relevance (distance) [Cong *et al.*, 2009; Li *et al.*, 2011; Zhang *et al.*, 2013; Chen *et al.*, 2018], discovering neighborhood patterns [Han and Wen, 2013; Han *et al.*, 2014; 2016], or processing location-based multimodal data joins (e.g., [Shang *et al.*, 2019; 2018; 2017]).

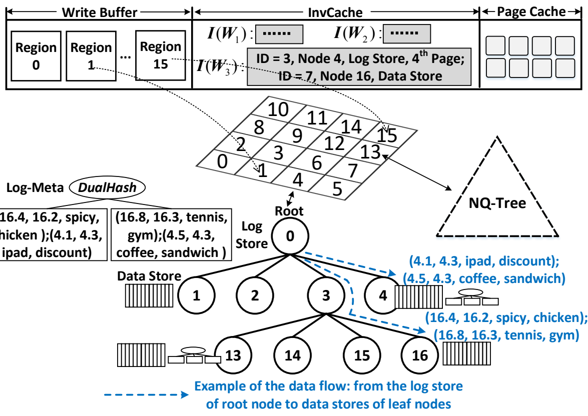

Write Buffer
InvCache
Page Cache
Region 0
Region 1
Region 15
$I(W_1)$: ......
$I(W_2)$: ......
$I(W_3)$:
ID = 3, Node 4, Log Store, 4th Page;
ID = 7, Node 16, Data Store
10
11
8
14
15
2
9
12
13
0
3
6
7
1
4
5
NQ-Tree
Log-Meta
DualHash
(16.4, 16.2, spicy, chicken);(4.1, 4.3, ipad, discount)
(16.8, 16.3, tennis, gym);(4.5, 4.3, coffee, sandwich)
Root
Log Store
0
(4.1, 4.3, ipad, discount);
(4.5, 4.3, coffee, sandwich)
Data Store
1
2
3
4
(16.4, 16.2, spicy, chicken);
(16.8, 16.3, tennis, gym)
13
14
15
16
Example of the data flow: from the log store of root node to data stores of leaf nodes

Figure 1: Overview of the NQ-tree

Data compression. Many techniques have been proposed for index compressions, e.g. [Anh and Moffat, 2005; 2006; Yan *et al.*, 2009; Pibiri *et al.*, 2019]. Due to space constraints, we outline the techniques used in our work, which have low preprocessing and running cost. S16 [Anh and Moffat, 2006] tries to pack as many values as possible into a 32-bit word with 16 ways of data organizations. NewPFD [Yan *et al.*, 2009] compresses a sequence of integers in a compact array by finding the smallest common bit-width b that can represent most of integers (say, 90%). Integers that cannot be represented in b bits will be stored as $exceptions$. Thereafter it stores the lower b bits of $exceptions$ in the compact array, and uses S16 to encode the higher $overflow$ bits separately.

3 The NQ-Tree Framework

3.1 Design Principles

We consider the following two principles in our design.

- $\mathcal{P}1$. *Efficient keyword and spatial pruning.* Combining both keyword and spatial information is more efficient for pruning than using each separately.
- $\mathcal{P}2$. *Navigating between update and search efficiency.* The index should be self-adjustable to the change of data distributions and access patterns, and thereby offering efficient support for both updates and searches.

3.2 Overview of NQ-Tree

An overview of the NQ-tree is shown in Fig. 1. After an initial coarse-grained geographic space partition, NQ-tree uses the Quadtree with low maintenance cost for deeper space decomposition [Kothuri *et al.*, 2002]. The difference is that NQ-tree uses a multi-store approach, each with different optimization purposes. Each node is attached with a write-friendly log store that buffers updates to the node and its descendants. Updates are first inserted into the write buffer, and over their life time flow into the compact data stores in leaf nodes in a cascading manner through log stores in all layers.

The multi-store feature faces two challenges. First, searches over the multiple stores should keep the read amplification (i.e., disk reads per search) as low as possible. Following the design principle $\mathcal{P}1$, a *log-meta* page is added to each log store, which uses space-efficient, high performance dual hash tables (Section 3.3) to store the summary spatial and keyword information. Moreover, the invCache caches the storing information of least frequent keywords, and is used to directly locate the candidate stores if a search contains keywords in it. Second, all stores must be carefully designed, composed well, and the transformation between them should be coordinated effectively. Following the design principle $\mathcal{P}2$, NQ-tree uses a self-tuning method that refines the index structure and leaf sizes adaptively to fit the workloads. It utilizes an cost model based online algorithm to address the following two issues.

- $\mathcal{I}1$. *What is the best time of emptying a node's log store?* For example, if a node goes through write-intensive workloads, we prefer emptying the log store as late as possible so that the empty cost can be amortized by more entries. While for read-intensive workloads, emptying the log store eagerly is more beneficial since the cost of checking the log store is saved for subsequent searches.
- $\mathcal{I}2$. *What is the optimal leaf size?* For instance, searches with large spatial regions expect big leaf nodes. However, an increased leaf size leads to higher write cost.

3.3 NQ-Tree Design

Log Store

For each object in the log store, we generate a 15-bit signature $\mathcal{S}$ for each of its keywords. The signature includes a 10-bit $hashTag$ of the keyword, a 2-bit $locTag$ indicating which child cell the object locates in, and a 3-bit $posTag$ showing which log page the object resides in.

A space efficient index is needed to accommodate signatures in the log-meta page. Cuckoo hashing [Pagh and Rodler, 2004] is a space efficient hash table which resolves collisions by kick-out evictions. However, it has two problems for signatures storing. One is that some keywords reside in multiple child cells and log pages, which means that there are 2^5 duplicate $hashTags$ in worst cases. This might lead to an infinite loop of kick-out evictions even when the load factor is low. The other one is that the insert performance deteriorates with the increasing load factor due to lots of kick-out evictions.

DualHash. We propose the DualHash that considers both space efficiency and performance. As Fig. 2 shows, it has two separate hash tables. The layout of main table is the same as the space efficient cuckoo hashing. The alternative table is a chained hash, in which buckets have to reserve space to maintain hash chains. The insert procedure first locates two positions in the main table, but find an empty slot in a linear probing manner without kick-out evictions. We restrict the probing length to 4 since a 4-way set associative cuckoo hashing has high space efficiency [Fountoulakis *et al.*, 2016]. Load balancing is considered when inserting in the main table. If a insertion fails in the main table, then it tries the alternative table. The false positive rate of a signature is $f \in [\frac{8}{2^{15}}, \frac{8+C_{altern}}{2^{15}}]$, where C_{altern} is the capacity of alternative table. The reason is that there are 8 possible positions in the main table for the signature, and C_{altern} possible positions in the alternative table in the worst case, i.e., the length of hash chain is C_{altern}.

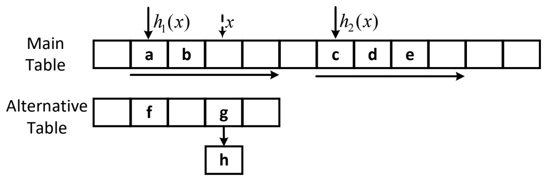


Figure 2: Structure of the *DualHash*

Maximize the capacity of log-meta page. The insert procedure can be simulated as throwing balls uniformly and independently into the main table's $\frac{C_{main}}{8}$ buckets, and each bucket can accommodate 8 balls at most. Once a bucket is full, balls arrived later are put in the alternative table. We use the Poisson distribution to model the number of balls in each bucket. Assume that the desired number of balls in each main table's bucket is μ, then the expected number of balls in the alternative table is:

$$E_{altern} = (\sum_{i=9}^{8+C_{altern}} \frac{\mu^i}{i} \cdot e^{-\mu} \cdot (i-8)) \cdot \frac{C_{main}}{8} \leq C_{altern} \quad (1)$$

where C_{main} and C_{altern} are the capacity of main table and alternative table. In addition, the size of the main table and alternative table should satisfy the page size (P) constraint:

$$C_{main} \cdot S_{main} + C_{altern} \cdot S_{altern} \leq P \quad (2)$$

where S_{main} and S_{altern} are the size of index entries. Now, the problem is to maximize the expected capacity:

$$E_{DualHash} = \frac{C_{main}}{8} \cdot \mu + C_{altern} \quad (3)$$

Given S_{main}, S_{altern}, P, and Equations(1-3), we can enumerate all the combinations of C_{main} and C_{altern} exhaustively, and find the optimal value.

Further improve the capacity of log store. We further improve the capacity of log store by eliminating signatures based on two types of keywords (denoted as *sig-skipping-keywords*):(1) *Least frequent keywords in invCache.* The summary information of these keywords based signatures can be found in the invCache. (2) *Keywords with a high probability of residing in the log page.* Given a keyword k, the probability of a log page containing it is $P(f_k) = 1 - (1 - f_k)^B$, where $f_k \in [0, 1]$ is the frequency distribution of the keyword and B is the average capacity of a log page. When generating signatures, we skip the most frequent keywords with poor pruning efficiency.

Data Store

Each leaf has a compact and static data store. As Fig. 3 shows, it combines data compression techniques and a space-efficient key-existence index to lower the write cost of merge operations and read amplification of searches.

Key-existence index. The data store takes advantage of the Bloom filter [Bloom, 1970], a space-efficient probabilistic data structure, to avoid accessing the inverted lists if it doesn't contain all the search keywords.

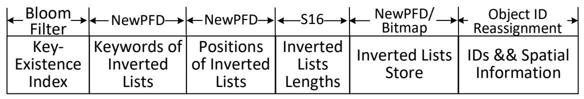


Figure 3: Layout of the data store

Data compression. As Fig. 3 shows, for sorted data, such as the keywords/positions/content of inverted lists, the *d-gap* based NewPFD scheme is used. It decreases the values that needs to be compressed by encoding the gaps (called *d-gaps*) between successive elements, which result in high compression ratio. For the data with high variances and relative small values, the S16 scheme with high performance is used, e.g., the length of inverted lists. We combine Bitmaps with object IDs reassignment to further improve the compression ratio. The reassignment procedure maps IDs of $\mathcal{N}_{data}$ objects into a new range $[1 : \mathcal{N}_{data}]$, which aims to reduce the *d-gaps* of inverted lists. Previous work [Blanco and Barreiro, 2005] has shown that minimizing the average *d-gaps* is NP-complete, and considering it as the traveling salesman problem (TSP) produces good results. Thus we use the greedy algorithm for TSP to generate new IDs. Thereafter, all the spatial information of objects are stored in an array by the pair $\langle o.ID, o.l\rangle$, where ID is the original object ID, and the position of the pair in the array indicates the new ID. We propose to use a hybrid method for inverted lists compression, in which a Bitmap with $\mathcal{N}_{data}$ bits or the NewPFD is used depending on their space efficiency. The value of jth bit in the Bitmap indicates whether the inverted list contains the new ID j. Given a *d-gap* based inverted list with length L_i, min value min_i, max value max_i, and mean value μ_i, according to Hoeffding's Inequality, we have the estimated bit-number b of NewPFD:

$$2^b \geq \mu_i + \sqrt{\frac{(max_i - min_i)^2}{2L_i}\log(\frac{2}{1-0.9})}$$

Next, for the *exceptions* of NewPFD, we assume their positions and values follow a uniform distribution, and use their expectations to estimate the space required by S16.

3.4 Operations on NQ-Tree

Update. Updates are initially inserted into the write buffer as logs. If the write buffer is full, the *batchInsertNode* routine (see Algorithm 1) is invoked to push updates to root nodes.

Search. If a search contains keywords in the invCache, it directly locates the candidate stores through list intersections. Then it invokes specific subroutines (see Algorithms 2 and 3) to collect matched objects. Otherwise, it searches the NQ-tree in a top-down manner, and checks all the encountered stores.

4 Self-Tuning Method

We first propose a cost model to evaluate the index refinements. With the cost model, the self-tuning method uses an online algorithm to adapt the index to workload shifts.

Algorithm 1 BatchInsertNode

```
   Input: $\mathcal{O}$: inserted objects; $\mathcal{N}$: node accessed currently.
1  foreach object $o \in \mathcal{O}$ do
2      Append $o$ to $\mathcal{N}$'s log store;
3      if $o.\psi$ contains keywords in the invCache then
4          Update the storing information of $o$ in the
           invCache;
5      end
6      Generate $o$'s signatures and update the log-meta
       page;
7  end
8  if Empty $\mathcal{N}$'s log store is required then
9      if $\mathcal{N}$ is an inner node then
10         if all $\mathcal{N}$'s children are leaf nodes and merging
           $\mathcal{N}$'s children is required then
11             Merge all $\mathcal{N}$'s children and set $\mathcal{N}$ as a leaf
               node;
12             return;
13         end
14         Push down $\mathcal{N}$'s objects recursively;
15     else /* $\mathcal{N}$ is a leaf node */
16         Integrate $\mathcal{N}$'s log store into the data store;
17         if a split of $\mathcal{N}$ is required then
18             Split $\mathcal{N}$;
19         end
20     end
21 end
```

4.1 Cost Model

Assume that the cost of disk seek is I_r, and the cost of transferring one page is I_s. Then we have the cost of accessing n random pages and n sequential pages as:

$$\mathcal{D}_r(n) = (I_r + I_s) \cdot n; \quad \mathcal{D}_s(n) = I_r + I_s \cdot n$$

Next, we compute the expected cost of emptying the log store. For inner nodes, we denote $\mathcal{N}^{C_i}_{log}$ as the number of objects pushed into its ith child. Assume that the log store has $P^{\mathcal{N}}_{log}$ pages and the average capacity of a log page is B, then we have the expected I/O cost as:

$$\mathcal{C}^{\mathcal{I}}_{empty} = \mathcal{D}_r(P^{\mathcal{N}}_{log}) + \sum_{i=0}^{3} sgn|\mathcal{N}^{C_i}_{log}| \cdot (\mathcal{D}_r(2) + \mathcal{D}_s(\lceil\frac{\mathcal{N}^{C_i}_{log}}{B}\rceil))$$

For leaf nodes, the number of objects in the log store and data store is denoted as $\mathcal{N}_{log}$ and $\mathcal{N}_{data}$, P is the page size, and $S^{\mathcal{N}}_{data}$ is the size of the data store. Then we have:

$$\mathcal{C}^{\mathcal{L}}_{empty} = \mathcal{D}_r(P^{\mathcal{N}}_{log}) + \mathcal{D}_s(\lceil\frac{S^{\mathcal{N}}_{data}}{P}\rceil) + \mathcal{D}_s(\lceil\frac{(\mathcal{N}_{log} + \mathcal{N}_{data}) \cdot S^{\mathcal{N}}_{data}}{\mathcal{N}_{data} \cdot P}\rceil)$$

Finally, we compute the expected cost of nodes splitting/merging. We denote $\mathcal{N}^{C_i}_{data}$ as the number of objects split into ith child. The cost of nodes splitting is the following:

$$\mathcal{C}_{split} = \mathcal{D}_r(P^{\mathcal{N}}_{log}) + \mathcal{D}_s(\lceil\frac{S^{\mathcal{N}}_{data}}{P}\rceil) + \sum_{i=0}^{3}\mathcal{D}_s(\lceil\frac{(\mathcal{N}^{C_i}_{log} + \mathcal{N}^{C_i}_{data}) \cdot S^{\mathcal{N}}_{data}}{\mathcal{N}_{data} \cdot P}\rceil)$$

Algorithm 2 SearchLogStore

```
Input: $q$: geo-textual search; $\mathcal{N}$: node accessed
       currently; $\mathcal{P}_{inv}$: Primary candidate log pages
       filtered by the invCache.
Output: $\mathcal{R}$: matched objects in $\mathcal{N}$'s log store
1  if all $q.\psi$ are sig-skipping-keywords then
2      if $q.\psi$ contains keywords in the invCache then
3          $\mathcal{P}_{vef} \leftarrow \mathcal{P}_{inv}$;
4      else
5          $\mathcal{P}_{vef} \leftarrow$ all $\mathcal{N}$'s log pages;
6      end
7  else
8      $\mathcal{H} \leftarrow$ generate $hashTags$ for $q.\psi$ except the
       sig-skipping-keywords;
9      foreach $hashTag$ $ht \in \mathcal{H}$ do
10         $\mathcal{S}_{hash} \leftarrow$ signatures in log-meta page that
           contain $ht$;
11         $\mathcal{S}_{loc} \leftarrow$ signatures in $\mathcal{S}_{hash}$ with the child cell
           indicated by $locTag$ overlaps with $q.r$;
12         Add $\mathcal{S}_{loc}$ to the list $\mathcal{L}[ht]$;
13     end
14     Intersect lists in $\mathcal{L}$ by signatures' $posTag$;
15     Add intersection results to $\mathcal{P}_{sig}$ as candidate log
       pages;
16     if $q.\psi$ contains keywords in invCache then
17         $\mathcal{P}_{vef} \leftarrow \mathcal{P}_{sig} \cap \mathcal{P}_{inv}$;
18     else
19         $\mathcal{P}_{vef} \leftarrow \mathcal{P}_{sig}$;
20     end
21 end
22 Search log pages in $\mathcal{P}_{vef}$ and add matched objects to $\mathcal{R}$;
```

The procedure of merging child nodes can be simulated as first empty the log store of each child C_i, and then write them as **one** compact data store. Thus we have:

$$\mathcal{C}_{merge} = \sum_{i=0}^{3} \mathcal{C}_{empty}^{C_i} - 3 \cdot I_r$$

.

4.2 Online Algorithm

We abstract the problem of online index refinements to a state transition problem. Depending on the existence of log stores, we associate each node with two states, i.e., s_{log} and s_{noLog}. Moreover, leaf nodes can stay in a third state s_{inner}. When a leaf node is transformed into the s_{inner} state, it is split into smaller leaf nodes. Analogously, for each inner node with leaf nodes as its children, we associate it with a third state s_{leaf}. When a specific inner node is set to the s_{leaf} state, it is transformed into a bigger leaf node by consolidating all its children. Now, the problem is how to schedule the states of each node that minimizes the overall cost incurred. A problem with similar flavor is the Metrical Task System problem [Borodin and El-Yaniv, 2005], and there exists an optimal deterministic online solution called *Work Function*.

Work function. Given a node $\mathcal{N}$, we refer to the cost of serving request σ_t at state s_i as $c^{\mathcal{N}}(s_i, \sigma_t)$, the cost of changing from state s_i to s_j as $d^{\mathcal{N}}(s_i, s_j)$. Let $w_t^{\mathcal{N}}(s_i)$ be the minimum cost to process a sequence of $t = \sigma_1 \cdots \sigma_t$ requests on $\mathcal{N}$ with an ending state s_i. Assume that $\mathcal{N}$ has served the requests sequence and that it is currently in state s_t. To process the next request σ_{t+1}, move to the state $s_{t+1} = s$ that minimizes $w_{t+1}^{\mathcal{N}}(s) + d^{\mathcal{N}}(s_t, s)$.

Algorithm 3 SearchDataStore

```
Input: $q$: geo-textual search; $\mathcal{N}$: node accessed
       currently.
Output: $\mathcal{R}$: matched objects in $\mathcal{N}$'s data store
1  if $q.\psi$ matches $\mathcal{N}$'s Bloom filter then
2      $\mathcal{L} \leftarrow$ Inverted lists for keywords in $q.\psi$;
3      $\mathcal{R}_{vef} \leftarrow$ Intersect the inverted lists;
4      foreach object $o \in \mathcal{R}_{vef}$ do
5          Add $o$ to $\mathcal{R}$ if $o.l \in q.r$;
6      end
7  end
```

Implementation. We use dynamic programming to maintain a set of state variables $w_t^{\mathcal{N}}(s_i)$ for each node. When a new request σ_t arrives, $w_t^{\mathcal{N}}(s_i)$ is updated as follows:

$$w_t^{\mathcal{N}}(s_i) = min\{w_{t-1}^{\mathcal{N}}(s_j) + d^{\mathcal{N}}(s_j, s_i) + min\{c^{\mathcal{N}}(s_i, \sigma_t), c^{\mathcal{N}}(s_j, \sigma_t)\} | s_j \in \{all\ of\ \mathcal{N}'s\ states\}\}$$

where $d^{\mathcal{N}}(s_j, s_i)$ is estimated by the cost model, and $c^{\mathcal{N}}(s_i, \sigma_t)$ is estimated by the characteristic of the request.

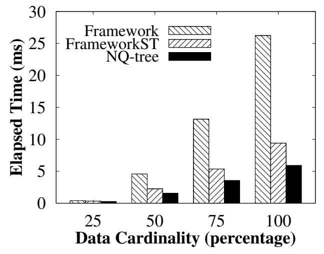
5 Experiment

5.1 Baselines

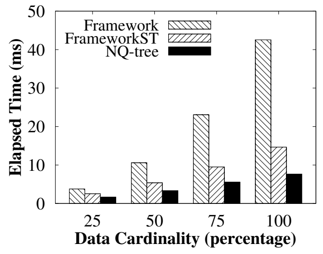
As described in Section 1, most of existing methods are based on the R-tree. However, the R-tree is inherently not suitable for update-intensive workloads [Biveinis *et al.*, 2007] because an update might modify multiple nodes due to the adjustment of overlapped MBRs. What's worse, methods based on it also require propagated updates on the embed inverted indexes. Together, it makes them impractical for stream processing. Others either rely on search logs and dataset analysis or global data compressions that make updates computationally prohibitive. Thus, we modify some existing indexes as baselines.

***Spatial-first* index (SPF).** A number of previous approaches [Zhang *et al.*, 2013; 2016; Wang *et al.*, 2015] have used the Quadtree due to its low maintenance cost, thus we use it as the *spatial-first* index. We set the page size as the maximum size of leaf nodes. The Quadtree only needs one page write for each update if there is no split operation. However, each geo-textual search has to check all the cells overlapped with the search region, which might need a large number of random reads, especially in dense data areas.

***Keyword-first* index (KWF).** We build inverted lists for all keywords to facilitate the *keyword-first* pruning. Each inverted list records the IDs of objects that contain the associated keyword, and is kept on the disk by one dimensional index (e.g., B+-Tree). For frequent keywords, the inverted lists

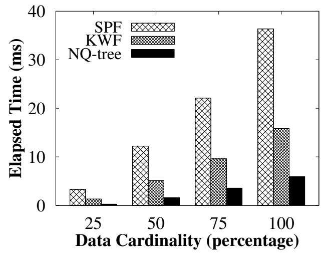


(a) 4SQ

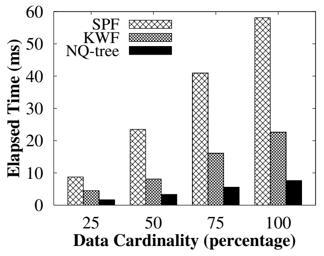


(b) TWEETS

Figure 4: Evaluation of proposed methods.



(a) 4SQ



(b) TWEETS

Figure 5: Performance comparison varying data cardinality.

are partitioned by coarse grained spatial regions. In addition, geo-locations of objects are indexed by IDs. The *keyword-first* search first identifies results by intersecting the inverted lists of search keywords in increasing order of frequencies, which allows skips on inverted lists. Then it conducts geo-location verifications on the returned results. The major advantage of KWF is that it consumes fast sequential disk accesses to scan inverted lists. However, each update consumes multiple expensive random writes on the corresponding inverted lists.

5.2 Experimental Setup

Environment. The experiments were ran on a workstation powered by Intel Xeon Gold-6148 CPU on Linux (Ubuntu 16.04), having a 15K RPM disk. All the experiments were conducted using the direct I/O mode to eliminate influences caused by the data caching of file systems.

Dataset. We experimented on two real-world datasets: 4SQ and TWEETS. The dataset 4SQ contains 4 million world-wide check-ins with both location and text information from Foursquare. The dataset TWEETS consists of 20 million geo-tagged tweets from users in the USA. We sampled 10% of the data as insertions/deletions, which can reflect the real-world data distributions, and the rest as basic data. We referred to the method in [Zhang *et al.*, 2016] that considers location skewness (dense areas are searched more often) and keyword likelihood to generate searches. The number of search keywords $|q.\psi|$ ranges from 1 to 5, and the search region $q.r$ ranges from 1km to 10km. By default, $|q.\psi|$ and $q.r$ are **3** and **5km**, and the ratio of searches (denoted as search ratio) is **0.5**. For updates, the ratio of insertions and deletions is **50%/50%**.

Index settings. We set the page size to 4 KB, and set the buffer size to 64MB and 256MB for 4SQ and TWEETS. An LRU buffer manager was implemented. In our design, we devoted part of the buffer as the write buffer and invCache. Specifically, 4MB memory was allocated for the write buffer so that the geo-space was initially divided into 1024 grid cells. The invCache cached the storing information of 40% least frequent keywords, accounting for no more than 5% of the total data. When generating signatures, we skipped the frequent keywords that have more than 50% probability of residing in a log page.

5.3 Evaluating Proposed Techniques

In this section, we evaluate the efficiency of our proposed methods. We implement the following methods. (1) Framework performs basic cascading updates with the log store and invCache designs. The capacity of data store is set to the estimated capacity of log store; (2) FrameworkST improves Framework with self-tuning methods; (3) NQ-tree integrates the hybrid compression method into FrameworkST.

We varied the data cardinality and reported the average elapsed time per operation. The results are shown in Fig. 4. We observed that FrameworkST outperformed Framework by up to 2.9× when the data cardinality was large ($\geq 75\%$). This is because FrameworkST continuously monitors the workload shifts, and dynamically refines its structure to reduce the overall access cost. For example, in FrameworkST, we observed that there were few log stores in nodes near the root node since these nodes saw lots more searches than batch updates, thus FrameworkST chosen to recursively push down updates to lower levels. Moreover, FrameworkST adapts the leaf size with the increase of data cardinality, which result in better scalability. We also observed that the hybrid compression method helped improve the performance. This is because it reduces the I/O cost of updates and searches on leaf nodes.

5.4 Comparing with Baselines

Scalability. Figure 5 depicts the result for scalability experiments. We can see that our approach showed good scalability. Note that the performance of SPF decreased faster than others with the increase of data cardinality, this is because the fixed leaf size is not adaptive to the datasets. For example, given a fixed search range, with the increase of data cardinality, the number of leaf nodes checked by searches might increase faster than linear since real-world datasets have skewed distributions. KWF scaled better than SPF, because it partitions the inverted lists of frequent keywords by spatial regions, which alleviate the side-effect caused by skewed datasets.

Effect of search ratio. Figure 6 shows the performance comparison on various search ratios. We can see that NQ-tree always achieved the best performance and improved the baselines up to 40×, especially in update intensive workloads. For example, on TWEETS dataset, when the search ratio was 0, the average elapsed time per update for SPF, KWF and NQ-tree were 1.095ms, 4.466ms, and 0.096ms, respectively. This is because NQ-tree takes advantage of the cascading update technique, which can amortize write cost over multiple

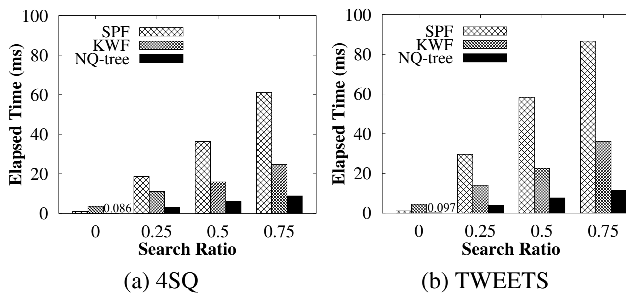


Figure 6: Performance comparison varying search ratio.

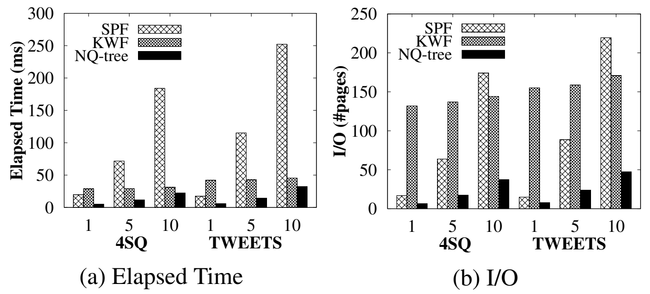


Figure 7: Performance comparison varying search range.

updates. We also found that NQ-tree scaled well when the search ratio was increased. The main reason is that NQ-tree can dynamically empty the log stores and adjust its leaf size to match the access patterns and data distributions, and it makes use of both location and keyword information for efficient pruning.

Effect of the search range. We also investigated the effect of scaling search regions. Figure 7 displays the elapsed time and I/O cost per search when scaling search regions. We observed that the performance of NQ-tree was rather steady. For example, on 4SQ dataset, when search region scaled from 1km to 10km, the average elapsed time for NQ-tree were 4.96/11.78/22.48 ms, while for the SPF the results were 19.67/71.70/184.12 ms. The main reason is that NQ-tree enlarges the size of leaf nodes when the search region is scaled, which benefits from fast sequential disk reads when searching these leaf nodes. Furthermore, enlarging the leaf nodes might also reduce the disk seek cost of loading Bloom filters that are used for pruning. Note that KWF also benefits from fast sequential accesses though it has a large number of disk reads.

Effect of the number of search keywords. Figure 8 displays the effect of the search keyword number. We observed that NQ-tree had the best performance in nearly all cases. This is because the well composed signatures, Bloom filters, invCache and Quadtree can utilize both keyword and spatial information to improve the pruning efficiency. The more search keywords, the better pruning efficiency. As expected, the performance of KWF degraded when the number of search keywords increased because it had to scan more inverted lists.

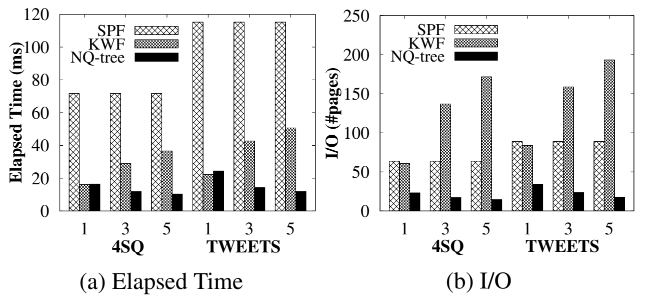


Figure 8: Performance comparison varying search keywords.

Effect of the insertion/deletion ratio. Generally, the index will shrink/grow if there're more deletions/insertions. The performance improved slightly when we increased the portion of deletions, which was consistent with the scalability results in Fig. 5. We didn't include the result due to space limit.

6 Conclusion

In this paper, we presented a novel index named NQ-tree for efficient geo-textual streaming data processing. In contrast to previous studies, which focused on search optimizations of static datasets, NQ-tree aims to offer efficient support for both updates and searches. We extended the Quadtree with multiple stores, which utilizes cascading updates to improve the update performance. For efficient searches, we designed space-efficient auxiliary structures to enhance the pruning capacity. Further, we proposed a self-tuning method to refine the index structure adaptively to fit the workload shifts. The experimental results demonstrate the efficiency of our proposal.

References

[Anh and Moffat, 2005] Vo Ngoc Anh and Alistair Moffat. Inverted index compression using word-aligned binary codes. *Information Retrieval*, 8(1):151–166, 2005.

[Anh and Moffat, 2006] Vo Ngoc Anh and Alistair Moffat. Improved word-aligned binary compression for text indexing. *IEEE Trans. Knowl. Data Eng.*, 18(6):857–861, 2006.

[Arge, 2003] Lars Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, 2003.

[Biveinis *et al.*, 2007] Laurynas Biveinis, Simonas Šaltenis, and Christian S Jensen. Main-memory operation buffering for efficient r-tree update. In *VLDB*, pages 591–602, 2007.

[Blanco and Barreiro, 2005] Roi Blanco and Alvaro Barreiro. Characterization of a simple case of the reassignment of document identifiers as a pattern sequencing problem. In *SIGIR*, volume 5, pages 587–588, 2005.

[Bloom, 1970] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[Borodin and El-Yaniv, 2005] Allan Borodin and Ran El-Yaniv. *Online computation and competitive analysis*. Cambridge university press, 2005.

[Chen and Shang, 2019] Lisi Chen and Shuo Shang. Region-based message exploration over spatio-temporal data streams. In *AAAI*, 2019.

[Chen *et al.*, 2013] Lisi Chen, Gao Cong, Christian S. Jensen, and Dingming Wu. Spatial keyword query processing: An experimental evaluation. *PVLDB*, 6(3):217–228, 2013.

[Chen *et al.*, 2014] Lisi Chen, Yan Cui, Gao Cong, and Xin Cao. SOPS: A system for efficient processing of spatial-keyword publish/subscribe. *PVLDB*, 7(13):1601–1604, 2014.

[Chen *et al.*, 2018] Lisi Chen, Shuo Shang, Zhiwei Zhang, Xin Cao, Christian S. Jensen, and Panos Kalnis. Location-aware top-k term publish/subscribe. In *ICDE*, pages 749–760, 2018.

[Christoforaki *et al.*, 2011] Maria Christoforaki, Jinru He, Constantinos Dimopoulos, Alexander Markowetz, and Torsten Suel. Text vs. space: efficient geo-search query processing. In *CIKM*, pages 423–432, 2011.

[Cong *et al.*, 2009] Gao Cong, Christian Jensen, and Dingming Wu. Efficient retrieval of the top-k most relevant spatial web objects. *PVLDB*, 2(1):337–348, 2009.

[Fountoulakis *et al.*, 2016] Nikolaos Fountoulakis, Megha Khosla, and Konstantinos Panagiotou. The multiple-orientability thresholds for random hypergraphs. *Combinatorics, Probability and Computing*, 25(6):870–908, 2016.

[Han and Wen, 2013] Jialong Han and Ji-Rong Wen. Mining frequent neighborhood patterns in a large labeled graph. In *CIKM*, pages 259–268, 2013.

[Han *et al.*, 2014] Jialong Han, Ji-Rong Wen, and Jian Pei. Within-network classification using radius-constrained neighborhood patterns. In *CIKM*, pages 1539–1548, 2014.

[Han *et al.*, 2016] Jialong Han, Kai Zheng, Aixin Sun, Shuo Shang, and Ji-Rong Wen. Discovering neighborhood pattern queries by sample answers in knowledge base. In *ICDE*, pages 1014–1025, 2016.

[Hariharan *et al.*, 2007] Ramaswamy Hariharan, Bijit Hore, Chen Li, and Sharad Mehrotra. Processing spatial-keyword (SK) queries in geographic information retrieval (GIR) systems. In *SSDBM*, pages 16–16, 2007.

[Hong *et al.*, 2012] Liangjie Hong, Amr Ahmed, Siva Gurumurthy, Alexander J Smola, and Kostas Tsioutsiouliklis. Discovering geographical topics in the twitter stream. In *WWW*, pages 769–778, 2012.

[Kothuri *et al.*, 2002] Ravi Kanth V Kothuri, Siva Ravada, and Daniel Abugov. Quadtree and R-tree indexes in oracle spatial: a comparison using GIS data. In *SIGMOD*, pages 546–557, 2002.

[Li *et al.*, 2011] Zhisheng Li, Ken Lee, Baihua Zheng, Wang-Chien Lee, Dik Lee, and Xufa Wang. IR-tree: An efficient index for geographic document search. *IEEE Trans. Knowl. Data Eng.*, 23(4):585–599, 2011.

[Liu *et al.*, 2013] Bin Liu, Yanjie Fu, Zijun Yao, and Hui Xiong. Learning geographical preferences for point-of-interest recommendation. In *KDD*, pages 1043–1051, 2013.

[Pagh and Rodler, 2004] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[Pibiri *et al.*, 2019] Giulio Ermanno Pibiri, Matthias Petri, and Alistair Moffat. Fast dictionary-based compression for inverted indexes. In *WSDM*, pages 6–14, 2019.

[Shang *et al.*, 2017] Shuo Shang, Lisi Chen, Zhewei Wei, Christian S. Jensen, Kai Zheng, and Panos Kalnis. Trajectory similarity join in spatial networks. *PVLDB*, 10(11):1178–1189, 2017.

[Shang *et al.*, 2018] Shuo Shang, Lisi Chen, Zhewei Wei, Christian S. Jensen, Kai Zheng, and Panos Kalnis. Parallel trajectory similarity joins in spatial networks. *VLDB J.*, 27(3):395–420, 2018.

[Shang *et al.*, 2019] Shuo Shang, Lisi Chen, Kai Zheng, Christian S. Jensen, Zhewei Wei, and Panos Kalnis. Parallel trajectory-to-location join. *IEEE Trans. Knowl. Data Eng.*, 31(6):1194–1207, 2019.

[Walther and Kaisser, 2013] Maximilian Walther and Michael Kaisser. Geo-spatial event detection in the twitter stream. In *ECIR*, pages 356–367, 2013.

[Wang *et al.*, 2015] Xiang Wang, Ying Zhang, Wenjie Zhang, Xuemin Lin, and Wei Wang. Ap-tree: Efficiently support continuous spatial-keyword queries over stream. In *ICDE*, pages 1107–1118, 2015.

[Yan *et al.*, 2009] Hao Yan, Shuai Ding, and Torsten Suel. Inverted index compression and query processing with optimized document ordering. In *WWW*, pages 401–410, 2009.

[Yin *et al.*, 2013] Hongzhi Yin, Yizhou Sun, Bin Cui, Zhiting Hu, and Ling Chen. Lcars: a location-content-aware recommender system. In *KDD*, pages 221–229, 2013.

[Zhang *et al.*, 2013] Dongxiang Zhang, Kian-Lee Tan, and Anthony Tung. Scalable top-k spatial keyword search. In *EDBT*, pages 359–370, 2013.

[Zhang *et al.*, 2016] Chengyuan Zhang, Ying Zhang, Wenjie Zhang, and Xuemin Lin. Inverted linear quadtree: Efficient top k spatial keyword search. *IEEE Trans. Knowl. Data Eng.*, 28(7):1706–1721, 2016.

[Zhao *et al.*, 2016] Kaiqi Zhao, Gao Cong, and Aixin Sun. Annotating points of interest with geo-tagged tweets. In *CIKM*, pages 417–426, 2016.