The VLDB Journal (2015) 24:823–848
DOI 10.1007/s00778-015-0403-4

REGULAR PAPER

AP-Tree: efficiently support location-aware Publish/Subscribe

Xiang Wang[1] · Ying Zhang[2] · Wenjie Zhang[1] · Xuemin Lin[1] · Wei Wang[1]

Received: 29 March 2015 / Revised: 3 August 2015 / Accepted: 9 September 2015 / Published online: 5 October 2015

Abstract We investigate the problem of efficiently supporting *location-aware* Publish/Subscribe (Pub/Sub for short), which is essential in many applications such as location-based recommendation and advertising, thanks to the proliferation of geo-equipped devices and the ensuing location-based social media applications. In a location-aware Pub/Sub system (e.g., an e-coupon system), subscribers can register their interest as *spatial-keyword* subscriptions (e.g., interest in nearby iphone discount); each incoming geo-textual message (e.g., geo-tagged e-coupon) will be delivered to all the *relevant* subscribers immediately. While there are several prior approaches aiming at providing efficient processing techniques for this problem, their approaches belong to *spatial-prioritized* indexing method which cannot well exploit the keyword distribution. In addition, their textual filtering techniques are built upon simple variants of traditional inverted indexes, which do not perform well for the textual constraint imposed by the problem. In this paper, we address the above limitations and provide a highly efficient solution based on a novel adaptive index, named AP-Tree. AP-Tree adaptively groups registered subscriptions using keyword and spatial partitions, guided by a cost model. AP-Tree also naturally indexes ordered keyword combinations. Furthermore, we show that our techniques can be extended to process *moving* spatial-keyword subscriptions, where subscribers can continuously update their locations. We present efficient algorithms to process both stationary and moving subscriptions, which can seamlessly and effectively integrate keyword and spatial partitions. Our extensive experiments demonstrate that AP-Tree and its variant AP$^+$-Tree can achieve up to an order of magnitude improvement on efficiency compared with prior state-of-the-art methods.

Keywords Publish/Subscribe system · Spatial-keyword Queries · Stream · Continuous monitoring · Moving queries

Electronic supplementary material The online version of this article (doi:10.1007/s00778-015-0403-4) contains supplementary material, which is available to authorized users.

✉ Xiang Wang
xiangw@cse.unsw.edu.au

Ying Zhang
ying.zhang@uts.edu.au

Wenjie Zhang
zhangw@cse.unsw.edu.au

Xuemin Lin
lxue@cse.unsw.edu.au

Wei Wang
weiw@cse.unsw.edu.au

1 School of Computer Science and Engineering, The University of New South Wales, Sydney, Australia

2 QCIS, University of Technology, Sydney, Australia

1 Introduction

Content-based Publish/Subscribe (Pub/Sub for short) system has attracted a lot of attention since the last decade [1,17,37,48]. Subscribers can register their interest as subscriptions, and publishers issue messages which need to be delivered to all the *relevant* subscribers in a real-time manner. Recently, due to the proliferation of *user-generated content* and geo-equipped devices, there is a vast amount of data with both spatial and textual information, referred to as *spatial-textual* data; they often come in a rapid streaming fashion in many important applications such as social networks (e.g., Facebook, FourSquare and Twitter) and location-based ser-

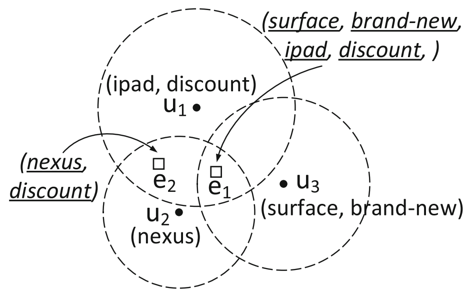


Fig. 1 Location-aware e-coupon system

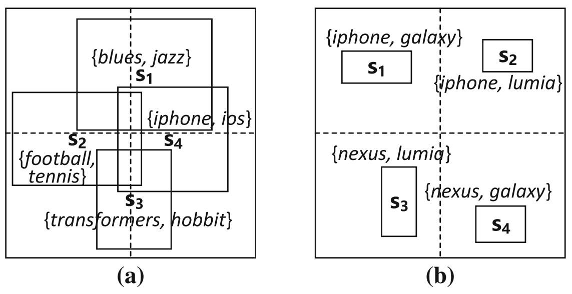


Fig. 2 Two motivating examples

vices (e.g., iAd[1]), thus leading to the *location-awareness* of subscribers. For instance, a user may want to retrieve all the tweets discussing the recent movie *Birdman* in her home city *Sydney*.

To make sense of streaming spatial-textual data and satisfy the increasing location-aware demand of subscribers, it is critical to develop efficient location-aware Pub/Sub system. In this paper, we investigate the problem of processing spatial-keyword subscriptions in a location-aware Pub/Sub system; that is, efficiently delivering a stream of incoming spatial-textual messages issued by publishers to all the *relevant* spatial-keyword subscriptions registered by a large amount of subscribers. This problem plays a fundamental role in a variety of applications such as information dissemination [45], location-based recommendation [33] and sponsored search [25].

Example 1 Figure 1 demonstrates a location-aware Pub/Sub system which delivers e-coupons to potential consumers. A user may register her interest as a subscription specified by a set of keywords and a spatial region. For instance, user u_1 wants to keep an eye on the discount ipad from *nearby* shopping malls and hence issues a subscription with keywords $\{ipad, discount\}$ and a circular region as shown in Fig. 1. Suppose two geo-tagged e-coupons e_1 and e_2 are released from two shops. Obviously, an e-coupon **matches** a user if the e-coupon's location is within the query range of her subscription, *and* all the search keywords are contained in the e-coupon. Therefore, in this example, e_1 will be delivered to $\{u_1, u_3\}$ and e_2 will be sent to $\{u_2\}$.

Challenges There are three key challenges in efficiently processing spatial-keyword subscriptions. *Firstly*, a massive number of subscriptions, typically in the order of millions, are registered in many applications, and hence even a small increase in efficiency results in significant savings. *Secondly*, the streaming spatial-textual messages (e.g., geo-tagged tweets) may continuously arrive in a rapid rate which also calls for high-throughput performance for better user satisfaction. *Thirdly*, novel techniques need to be created to develop spatial-textual indexing mechanism that adapts to both the spatial and keyword distributions of the subscription workload. To the best of our knowledge, [6] and [28] are the only two existing work that systematically study the same problem as ours. Two indexing techniques, IQ-Tree and R^t-Tree, are proposed to match each incoming message to relevant subscriptions following the *filtering-and-refinement* paradigm. Although a large number of irrelevant subscriptions can be pruned by IQ-Tree and R^t-Tree, they suffer from two fundamental drawbacks.

Firstly, the spatial factor is always prioritized during the index construction regardless of the keyword distribution of the subscriptions. One of our key observations is that the filtering powers based on the spatial and textual constraints may differ substantially under different subscription workload. Hence, an indexing method must *adapt* to *both* spatial and keyword distributions of the subscription set to achieve high efficiency. For example, in Fig. 2a, textual filtering is more effective because the query ranges of subscriptions are heavily overlapped while they can be easily distinguished by their keywords. On the contrary, we prefer spatial filtering in Fig. 2b since their ranges are scattered evenly throughout the space while their keywords are quite similar.

Secondly, the inverted indexing technique adopted in [6, 28] is not well suited to textual filtering given the nature of the problem is a *superset containment search* [21] from textual perspective. Although inverted indexes have been widely employed in conventional spatial-keyword queries, they are essentially designed for *subset containment search* [21], where a set of indexed objects *containing* all query keywords are retrieved. We observe that index structures specifically designed for superset containment search, such as the *ordered keyword trie* [22], shall offer better performance by exploiting the order of keywords, and indexing multiple keyword combinations.

Based on the above observations, we propose a novel index technique, namely the Adaptive spatial-textual Partition Tree (AP-Tree for short), to effectively organize spatial-keyword subscriptions. Besides, in order to satisfy the high-throughput and low-latency requirements of real-time dissemination, we follow most of the existing Pub/Sub

[1] http://advertising.apple.com.

systems (e.g., [7,20,23,28,35,41,48]) to implement all our indexes in memory. In a nutshell, AP-Tree is a f-ary *in-memory* tree structure where subscriptions are recursively divided by *spatial* or *keyword* partitions (nodes). A cost model is devised to rigorously guide the selection of partition methods such that the construction of the index is adaptive to the subscription workload. Moreover, we seamlessly and effectively integrate a variant of ordered keyword trie structure [22] to enhance the textual filtering performance.

On the other hand, depending on users' demands, some applications need to process moving subscriptions. For example, in the iAd, a famous mobile advertising platform developed by Apple Inc, recommendations (e.g., apps from App Store) may be pushed to user's iPhone when she walks in the city. Thus, it is desirable to extend our system to support moving spatial-keyword subscriptions.

There are three main challenges to support moving subscriptions. *Firstly*, it is desirable to design indexing structure which can support the continuous reorganization of moving subscriptions efficiently. The AP-Tree cannot work very well for moving subscriptions because the continuous movement of subscriptions will deteriorate the original structure of AP-Tree and trigger indexing maintenance. *Secondly*, it is crucial to design an index which can support *incremental* re-evaluation of moving subscriptions, instead of simply re-evaluating from scratch for each location update. *Thirdly*, the efficient organization of all the *active* messages using spatial-keyword indexing structure should also be considered so as to support efficient re-evaluation of moving subscriptions.[2]

To this end, we propose a new cost model to accommodate the moving subscriptions, following the same framework as AP-Tree. Besides, we integrate all the active messages into AP-Tree seamlessly and naturally, to support efficient incremental monitoring/re-evaluation when subscriber moves.

Our principal contributions are summarized as follows:

For stationary subscriptions

- We devise a novel adaptive spatial-textual partition tree (i.e., AP-Tree) to tackle spatial-keyword subscriptions over streaming spatial-textual messages. To the best of our knowledge, this is the first spatial-textual indexing mechanism which adaptively prioritizes spatial and keyword partition methods.
- A cost model is proposed to evaluate the goodness of *keyword partition* and *spatial partition*. For keyword partition, an optimal algorithm and an efficient heuristic algorithm are devised. As to spatial partition, we show that finding optimal spatial partition is *NP-hard* and propose an efficient heuristic algorithm instead. With the guide of cost model, AP-Tree is constructed in an adaptive way to minimize overall matching cost. Moreover, we show that AP-Tree is self-adjustable to the change of subscription workload.
- Comprehensive experiments show that our AP-Tree achieves substantial improvements (up to an order of magnitude speed up) over the state-of-the-art techniques. For instance, with 20 million registered subscriptions, our method can process around 2, 500 tweets per second, compared with about 300 tweets by the previous methods.

For moving subscriptions

- We propose a new indexing structure, namely AP$^+$-Tree, which is a natural extension of AP-Tree, by introducing a new cost model to accommodate the movement of subscriptions. AP$^+$-Tree considers not only the spatial and keyword distributions of subscriptions, but also the movement of subscribers.
- We propose to combine both subscription index and message index into a unified framework to support the efficient re-evaluation of moving subscriptions.
- Comprehensive experiments indicate that our AP$^+$-Tree is up to an order of magnitude faster over the baseline algorithms.

Roadmap This paper is an extension of our previous work on stationary spatial-keyword subscriptions [40]. In this extended paper, we extend techniques in [40] to support moving spatial-keyword subscriptions and present comprehensive experimental results to evaluate the efficiency and effectiveness of proposed techniques.

The rest of this paper is organized as follows. We introduce related work in Sect. 2. Section 3 presents techniques to process stationary spatial-keyword subscriptions, including the related algorithms and a cost model to guide the adaptive indexing construction. Section 4 presents the extension toward moving spatial-keyword subscriptions, including a unified indexing structure and a cost model variant to support the indexing of moving subscriptions. Extensive experiments of both stationary and moving subscriptions are depicted in Sect. 5. Finally, Sect. 6 concludes the paper.

2 Related work

In this section, we briefly review the literatures which are closely related to our problem.

Spatial-keyword Search In recent years, spatial-keyword search has attracted great attention, which aims to retrieve the *relevant* spatial-textual objects for a given spatial-keyword query. Existing work usually combine keyword indexing and spatial indexing techniques to organize objects such that non-

[2] Active messages refer to the messages which are currently alive and not expired in the system.

promising objects can be quickly pruned from both spatial and textual perspectives. In general, these techniques can be classified into two categories: *keyword-prioritized* (e.g., [10, 34,47,50]) and *spatial-prioritized* (e.g., [11,13,50]). We say an index is *keyword-prioritized* in the sense that it prefers keyword feature during index construction; similarly, we say an index is *spatial-prioritized* if it prefers spatial feature during index construction. Note that a spatial-keyword search is an *ad hoc/snapshot* query (i.e., user-initiated model), while our problem focuses on *continuous* subscriptions (i.e., server-initiated model). Please refer to [8] for a good review and summary of *ad hoc* queries.

Publish/Subscribe system In a Pub/Sub system, there are many long-running subscriptions registered by subscribers on the server side. The messages are coming in a stream fashion and need to be evaluated and reported to the *relevant* subscriptions. Most of the existing Pub/Sub systems investigate either content-based/predicate-based matching (e.g., [16,35,41,48]) or similarity-based ranking (e.g., [30, 37]). Nevertheless, they do not consider the spatial information. There are some existing work on the location-aware Pub/Sub systems, but most of them either cannot properly handle large-scale streaming data (e.g., [9]) or do not consider the textual information (e.g., [2]).

Recently, spatial-keyword Pub/Sub systems have been proposed by a line of researches [6,7,23,28]. They all aim to organize a large number of subscriptions to facilitate the dissemination of streaming spatial-textual messages. Among them, [6,28] focus on the exact same problem as ours (i.e., boolean range matching), while [7,23] tackle another line of problem (i.e., similarity searching). Thus, in this paper, we only consider the algorithms proposed in [6,28] as our competitors. Specifically, two efficient indexing techniques, namely IQ-Tree and R^t-Tree, are proposed to index a massive amount of subscriptions efficiently. Both of them belong to *spatial-prioritized* indexing mechanism where spatial feature is preferred during index construction. For the coherence of this paper, we delay the detailed discussions of IQ-Tree and R^t-Tree to Sect. 3.1.

Moving spatial-keyword query There are several work focusing on efficient processing of moving spatial-keyword queries. Wu et al. [42] and Huang et al. [24] aim to continuously monitor top-k results for a moving spatial-keyword query by utilizing *safe zone* technique. Safe zone is a region in which the query results keep constant. However, the naive safe zone technique [5,24,27,32,42,49] cannot be trivially adopted in our problem because the frequent incoming messages will destroy the pre-computed safe zone, thus forcing the repeated re-computation of safe zone. Besides, they only consider one query each time. Recently, Du et al. [14] and Guo et al. [19] solve the moving spatial-keyword query on road networks, by utilizing incremental evaluation strategy to reduce repetitive traversing of network edges. However, they only deal with static data. Another related work is *MobiFeed* proposed by Xu et al. [44]. Their work focuses on efficient *n-look-ahead* news feed schedule based on a location predicator to maximize total relevance score for all n timestamps, which is inherently different from our problem. Moreover, they do not consider indexing queries to match incoming messages.

To the best of our knowledge, a recent work by Guo et al. [20] is the only one that considers multiple moving spatial-keyword subscriptions against a stream of spatial-textual messages. Particularly, they propose a framework, called Elaps, which aims to reduce the *communication cost* between user clients and server by utilizing safe region and impact region techniques which are carefully designed for streams. A BEQ-Tree structure is also proposed to index messages. One main difference between their work and ours is that, we follow a different line of work [29,43] where the users need to report the location update periodically to the central server, and the matching results are delivered to the users in an incremental and periodical manner. Our main concern is to reduce the *computation cost* on the server by utilizing efficient indexing structures for both subscriptions and messages. Even if Elaps can reduce communication cost, it suffers from high memory cost and high server computation overhead, which is mainly incurred by safe region construction and BEQ-Tree matching. Thus, Elaps cannot scale to large number of subscriptions, while the scalability of subscriptions is one of the key challenges in our problem.

3 Stationary Publish/Subscribe

In this section, we deal with matching stationary subscriptions against a stream of incoming messages. We first give some preliminaries in Sect. 3.1. Then we present a meticulous introduction to the framework of AP-Tree in Sect. 3.2, which includes the detailed structure of AP-Tree and message matching algorithm. In Sect. 3.3, we propose a cost model to guide the construction and maintenance of AP-Tree. Finally, we end up with some discussions in Sect. 3.4.

3.1 Preliminaries

In this paper, $\mathcal{M}$ denotes a sequence of streaming spatial-textual messages. A spatial-textual message is a textual message with geo-location, such as check-ins and geo-tagged tweets. Formally, a spatial-textual message m is modeled as $m = (\psi, loc)$, where $m.\psi$ denotes a set of distinct terms (keywords) from a vocabulary set $\mathcal{V}$ and $m.loc$ represents a geo-location.[3]

[3] We assume the location of message is a point while our techniques can be immediately extended to support a spatial region, e.g., circle, rectangle.

Table 1 Summary of notations

Notation	Definition
m	A spatial-textual message
s	A spatial-keyword subscription
$m.\psi$ $(s.\psi)$	A set of keywords for m (s)
$m.loc$ $(s.r)$	Message location (subscription range)
w, w_i, w_j	Keyword/term
$\mathcal{S}$ (S)	Subscription set (subset of $\mathcal{S}$)
$\mathcal{M}$ (M)	Message stream (subset of $\mathcal{M}$)
$\mathcal{V}$ (V)	Vocabulary (subset of $\mathcal{V}$)
N	A node of AP-Tree
N_l	Offset of node N
N_r	Spatial region of node N
f	Fanout of AP-Tree node
θ_p	Partition termination threshold
θ_{KL}	KL-Divergence threshold

A spatial-keyword subscription s is defined as $s = (\psi, r)$, where $s.\psi$ is a set of distinct user-specified keywords and $s.r$ is a rectangle. Note that as we employ a recursive decomposition policy for space, other spatial regions such as polygon or circle can also be easily supported in our indexing structure [36]. A spatial-keyword subscription is a *continuous* long-running query, and is valid until it is unregistered by its subscriber.

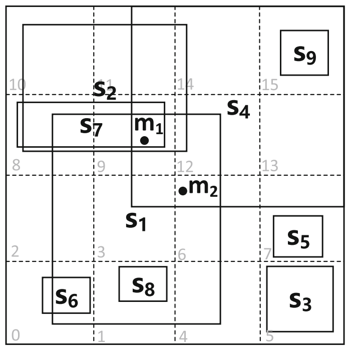
We say an incoming spatial-textual message *matches* (or *satisfies*) a spatial-keyword subscription if it satisfies both spatial and keyword constraints of the subscription. Following is a formal definition.

Definition 1 (*Matching*) A spatial-textual message *matches* a spatial-keyword subscription if and only if the following two conditions are satisfied: (1) $m.\psi \supseteq s.\psi$, and (2) $m.loc \in s.r$.

We also say a spatial-keyword subscription *matches* a spatial-textual message without ambiguity. Table 1 summarizes the mathematical notations used throughout this paper.

Problem statement We tackle the problem of matching stationary spatial-keyword subscriptions against streaming spatial-textual messages. Specifically, given a set $\mathcal{S}$ of subscriptions, for each incoming message m from streaming spatial-textual data $\mathcal{M}$, we aim to rapidly deliver m to all the matching subscriptions.

Example 2 Figure 3 depicts a running example used throughout this paper. In this example, there are 9 registered subscriptions $\{s_1, \ldots, s_9\}$ and two recent incoming messages $\{m_1, m_2\}$. Specifically, m_1 falls in the search ranges of $\{s_1, s_2, s_4, s_7\}$, and its keywords only fully contain all the keywords of s_7. Thus, m_1 is delivered to $\{s_7\}$. With similar rationale, m_2 matches subscriptions $\{s_1, s_4\}$.



s	ψ
s_1	w_1,w_2
s_2	w_1,w_2
s_3	w_2,w_4
s_4	w_1,w_5
s_5	w_1,w_3
s_6	w_3,w_5
s_7	w_3,w_4
s_8	w_4,w_6
s_9	w_5,w_6

$m_1{:}\psi = \{w_2, w_3, w_4\}$
$m_2{:}\psi = \{w_1, w_2, w_5\}$

Fig. 3 Running example

In the paper hereafter, we abbreviate the *spatial-textual message* and the *spatial-keyword subscription* as *message* and *subscription*, respectively, if there is no ambiguity. We assume there is a total order for keywords in $\mathcal{V}$, and the keywords in each subscription and message are sorted accordingly. For presentation simplicity, we assume $w_i < w_j$ if $i < j$.

In the following, we introduce the details of two state-of-the-art methods, i.e., IQ-Tree and R^t-Tree, which can support spatial-keyword subscriptions, and discuss some related techniques for our indexing design.

Previous state-of-the-arts In IQ-Tree [6], subscriptions are organized by a Quadtree where each subscription is attached to one or multiple Quadtree cells according to a cost model which aims to balance matching and update costs. For each cell, the related subscriptions are organized by a ranked-key Inverted List [46], and a subscription is assigned to the posting list of its least frequent keyword. Figure 4a shows an example of IQ-Tree where 9 subscriptions in Fig. 3 are organized. In particular, subscription s_7 in cell 9 is in the posting list of w_4 since w_4 is the least frequent keyword among $s_7.\psi = \{w_3, w_4\}$. The matching algorithm of IQ-Tree follows the *filtering-and-refinement* paradigm. For instance, regarding the incoming message m_1 in Fig. 3, unpromising subscriptions are first pruned based on their search ranges, i.e., only subscriptions which reside on the cells penetrated by m_1 (gray cells) survive the spatial filtering. Then keyword filtering is applied, and only the subscriptions on the posting lists of the message keywords are retrieved, which correspond to $\{s_1, s_2, s_7\}$. Finally, candidate subscriptions are verified based on their search ranges and keywords, and message m_1 is delivered to subscription $\{s_7\}$. The total number of subscriptions verified in this example is 3.

Regarding R^t-Tree [28], subscriptions are indexed by an R-Tree based on their search ranges. Each R-Tree node also records the keywords of its descendant subscriptions, namely *token filter*, for textual filtering purpose. Two variants of R^t-Tree, namely R^{t+}-Tree and R^{t++}-Tree, further improve the performance by carefully choosing one and

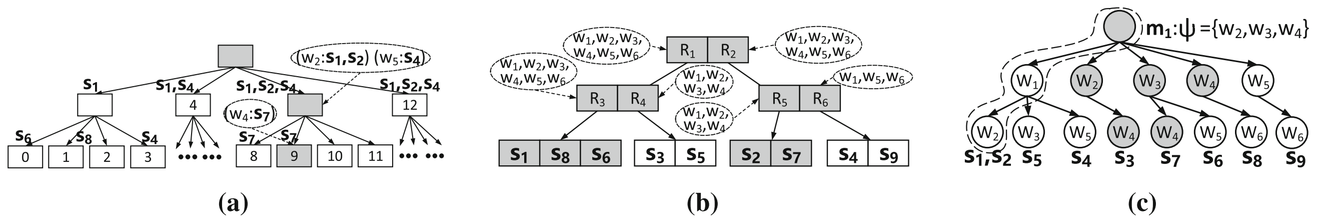


Fig. 4 Example of IQ-Tree, R^t-Tree and ordered keyword trie. **a** IQ-Tree. **b** R^t-Tree. **c** ordered keyword trie

multiple representative tokens (keywords), respectively. Figure 4b demonstrates an example of R^t-Tree. At each node, it employs both spatial and keyword filtering techniques to prune unpromising subscriptions. For example, to match message m_1 in Fig. 3, we need to access all the gray nodes and verify $\{s_1, s_8, s_6, s_2, s_7\}$ in the leaf nodes according to spatial and keyword constraints. Note that R_6 is pruned because its token set, i.e., $\{w_1, w_5, w_6\}$, has no overlap with keywords in m_1, i.e., $\{w_2, w_3, w_4\}$, while R_4 is pruned by spatial constraint. The total number of subscriptions verified in this example is 5.

Superset containment search The problem of *superset containment search* has been extensively studied in literatures, and many efficient techniques are proposed (e.g., [22,25,39]). Specifically, given a set of subscriptions and a message, each of which consisting of a set of keywords, we aim to find all the subscriptions whose keywords are fully contained by the message. Clearly, the nature of our problem is a superset containment search if the spatial dimension is not considered.

To efficiently support *superset containment search*, Zeinab et al. [22] recently propose an *ordered keyword trie* structure where each node corresponds to a keyword assuming that there is a global order for all keywords. Each subscription is indexed based on its ordered keywords (i.e., "prefixes"). Figure 4c depicts the ordered keyword trie structure for subscriptions in Fig. 3 where each subscription can be accessed through a unique path following its ordered keywords. For instance, s_1 with keywords $\{w_1, w_2\}$ can be visited through the path as indicated by the dotted polygon. Given message m_1 with $m_1.\psi = \{w_2, w_3, w_4\}$, we only need to visit gray nodes in Fig. 4c and come up with final matches $\{s_3, s_7\}$ w.r.t. keywords only.

In this paper, we integrate a variant of the ordered keyword trie structure in AP-Tree to efficiently support textual filtering.

3.2 AP-Tree framework

In this section, we present a novel adaptive spatial-textual indexing mechanism to organize subscriptions, namely AP-Tree (**A**daptive **P**artition Tree). Section 3.2.1 introduces the motivation of AP-Tree. Section 3.2.2 describes the AP-Tree structure, followed by a detailed matching algorithm in Sect. 3.2.3.

3.2.1 Motivation

Due to the massive number of subscriptions, it is imperative to devise efficient indexing technique such that a large number of unpromising subscriptions can be filtered at a cheap cost. We show that a good indexing mechanism over spatial-keyword subscriptions should satisfy following three criteria. (**1**) **Adaptiveness** Intuitively, with respect to different keyword and location distributions of the subscription workload, both spatial feature and textual feature may become the dominant factor. This observation is illustrated in Fig. 2 and substantiated by our empirical study. As shown in Sect. 3.1, tree structure of IQ-Tree [6] and R^t-Tree [28] is only determined by the spatial feature. Although the keyword filtering component (e.g., local Inverted List) is augmented to tree nodes, their overall performance is unavoidably deteriorated. On the other hand, our experiments show that *keyword-prioritized* indexing approach (e.g., RQ-Tree [4]) also suffers from the same problem. For example, our index is up to one order of magnitude faster than RQ-Tree in the experiments. This motivates us to devise a novel textual and spatial partition-based f-ary tree structure so that the subscriptions are indexed in an adaptive way w.r.t. the subscription workload. Moreover, the index should be self-adjustable to the change of subscription workload. In particular, two types of partition strategies, namely *keyword partition* and *spatial partition*, are proposed to recursively partition a set of subscriptions by textual feature and spatial feature, respectively. *A node partitioned by keyword (resp. spatial) feature is called* k-node *(resp.* s-node*) in our indexing structure*. A cost model (Sect. 3.3.1) is developed to decide which partition approach is employed at each node.
(**2**) **Efficient keyword filtering** From textual perspective, our problem is essentially a *superset containment search*; that is, finding subscriptions whose keywords are *fully contained* by a given message. Among existing techniques (e.g., [22,25,39]), ordered keyword trie [22] demonstrates

[4] RQ-Tree is a *keyword-prioritized* indexing structure derived from IQ-Tree and compared in our experiments.

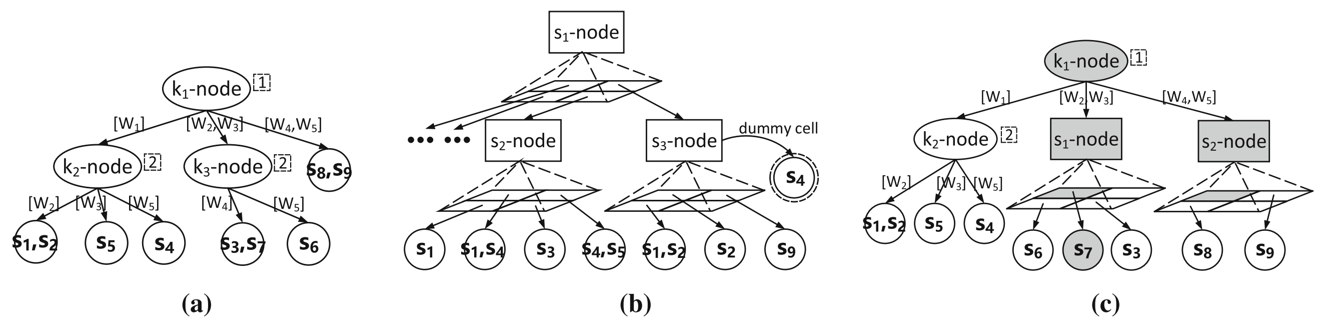


Fig. 5 Examples of keyword partition, spatial partition and AP-Tree. **a** Keyword partition, **b** spatial partition, **c** AP-Tree

its superior performance because it takes great advantage of common prefixes of the ordered subscription keywords. Moreover, it is a hierarchical tree structure. This motivates us to integrate the ordered keyword trie for keyword filtering purpose. To accommodate the f-ary tree structure of AP-Tree, we partition related keywords on the tree node into f parts based on our cost model, instead of keeping each individual keyword.[5] Optimal and heuristic keyword partition methods are proposed in Sect. 3.3.2.

(**3**) **Efficient spatial filtering** Regarding the spatial filtering, our problem corresponds to the *point stabbing search* [12] in two-dimensional space; that is, identifying subscription rectangles which are stabbed by the geo-location of the incoming message. The best-known data structure for the point stabbing problem is the segment tree [3] which can retrieve all k related rectangles with search time $O(\log n + k)$ where n is the number of subscriptions. However, segment tree is not well suited to large-scale data because of the space usage of $O(n \log n)$ on two-dimensional data. *Space-oriented* (e.g., Quadtree) and *object-oriented* (e.g., R-Tree) partition strategies are adopted in [6] and [28], respectively, due to their good support of point stabbing search and scalability. As stressed in [6], *space-oriented* partition strategy is more suitable to spatial filtering because of its disjoint space decomposition policy and good support of subscription ranges with different sizes. Specifically, *space-oriented* partition strategy can divide the subscription range into finer granularities, while *object-oriented* partition strategy has to maintain a single region for each subscription. Motivated by this, we adopt *space-oriented* partition approach for spatial partition. In particular, the region of each s-node in our indexing structure is partitioned into f grid cells guided by the cost model. As it is an NP-hard problem to find optimal spatial partition, an efficient heuristic algorithm is designed in Sect. 3.3.3.

3.2.2 AP-Tree structure

Based on the above motivations, we devise an adaptive spatial-textual partition tree (i.e., AP-Tree) which employs *keyword partition* and *spatial partition* methods to recursively divide subscriptions in a top-down manner. In this paper, N denotes an AP-Tree node and there are three types of nodes: *keyword node* (k-node), *spatial node* (s-node) and *leaf node* (l-node). An intermediate node is a k-node (resp. s-node) if *keyword partition* (resp. *spatial partition*) is adopted. We use f to denote the fanout of the intermediate node. Each subscription will be assigned to one or multiple l-nodes according to its subscription range and ordered keywords. The subscriptions in each l-node are organized as a map structure, namely s-list, to support efficient insertion and deletion.[6] Below, we introduce k-node and s-node in detail.

K-Node We assume there is a total order among keywords in the vocabulary $\mathcal{V}$, and keywords in each message and subscription are sorted accordingly. Subscriptions assigned to a node N are partitioned into f *ordered cuts* according to their N_lth keywords, where N_l is called the *partition offset* of the node N. We have $N_l < N_l^*$ if N^* is a descendant k-node of N. An *ordered cut* is an interval of the ordered keywords, denoted as $c[w_i, w_j]$, where w_i and w_j ($w_i \leq w_j$) are boundary keywords. For presentation simplicity, we use $c[w_i]$ to denote $c[w_i, w_i]$ if there is only one keyword in the cut.

Example 3 Figure 5a shows a special case of AP-Tree in which only keyword partition is employed on the running example. We use an *oval* to represent a k-node and the number on its right side indicates the partition offset. Meanwhile, a l-node is denoted by a *circle*, in which a list of subscriptions is stored (i.e., s-list). Assume there are at most 3 ordered cuts on each k-node. In k$_1$-node with partition offset 1, we collect the *first* keywords of 9 subscriptions

[5] Note that there are about 1.7 million distinct keywords in the tweet dataset.

[6] We should not be confused with the same "s" in s-list and s-node. s-list refers to *subscription* list, while s-node refers to *spatial* node. There is no relationship between them.

which correspond to $\{w_1, w_2, w_3, w_4, w_5\}$. These keywords can be divided into 3 cuts: $c[w_1]$, $c[w_2, w_3]$ and $c[w_4, w_5]$. Subscriptions $\{s_1, s_2, s_4, s_5\}$ are assigned to $c[w_1]$ whose corresponding node is k_2-node. Since the partition offset of k_2-node is 2, the *second* keywords of these subscriptions, i.e., $\{w_2, w_3, w_5\}$, are used to assign subscriptions into three cuts: $c[w_2]$, $c[w_3]$ and $c[w_5]$, each of which is associated with a l-node.

S-Node The space is recursively partitioned by s-nodes. Let N_r denote the region of a s-node N, which will be divided into f grid cells. A subscription on a s-node N is pushed to a grid cell c if $s.r$ overlaps or contains c. Note that, unlike the k-node in which a subscription is assigned to a unique cut, a s-node may assign a subscription to multiple cells.

Example 4 Figure 5b depicts another special case of AP-Tree in which only spatial partition is employed on the running example.[7] Here, we use a *rectangle* to represent a s-node. In each s-node, the spatial region is partitioned into 4 cells. To match a message, we simply navigate through the s-nodes which contain the message location, until we reach the l-node. We remark that the cells in a s-node may not be of equal area (i.e., the s-node may not be uniformly partitioned).

For each k-node N, a subscription s assigned to N cannot find a cut if there is no enough subscription keywords, i.e., $|s.\psi| < N_l$. We use a *dummy cut* to keep these subscriptions. Similarly, each s-node N has a *dummy cell* for the subscriptions which contain the region of N (i.e., $N_r \subseteq s.r$) and hence do not need to be further partitioned on node N. Note that subscriptions on the dummy cut (resp. cell) may be further partitioned by s-node (resp. k-node) only, or simply maintained by a l-node. For instance, the node indicated by dotted circle in Fig. 5b is actually a dummy node, because the query range of s_4 fully contains the region of s_3-node.

Example 5 Figure 5c illustrates an example of AP-Tree constructed over the running example, where both keyword and spatial partitions are employed. Subscriptions are recursively partitioned by k-node or s-node and finally assigned to l-node.

3.2.3 Incoming message matching

In this section, we present efficient algorithm for *incoming message matching*. Following the *filtering-and-verification* paradigm, we navigate through AP-Tree to prune non-promising subscriptions by utilizing spatial or keyword filtering techniques and then verify the candidate subscriptions on l-nodes accessed.

[7] For sake of clarity, we remind that the subscript in each s-node (e.g., s_1-node) in the figure has nothing to do with subscription s (e.g., s_1).

Algorithm 1 depicts the procedure to retrieve all the matching subscriptions for a given message m. It is a recursive procedure invoked by each accessed intermediate node with a depth-first search strategy. In particular, we simply verify the associated subscriptions if a l-node is accessed, and matching subscriptions are kept in $\mathcal{R}$ (Line 2). Regarding s-node (Lines 12–15), we only need to access the cell c stabbed by m (i.e., $m.loc \in c_r$) as well as the dummy cell. Recall that the dummy cell of a s-node keeps subscriptions covering the region of the node, and may be further partitioned by k-node only. As to the k-node (Lines 5–10), let $w_1, w_2, \ldots, w_{|m.\psi|}$ denote all the message keywords in $m.\psi$. For each k-node N accessed, we use η to denote the start matching position regarding the message keywords. Line 6 identifies the corresponding cut for each message keyword w_j ($\eta \le j \le |m.\psi|$). For each cut hit by at least one message keyword, we further explore its corresponding node at Line 8 where η is set to $i + 1$ and w_i denotes the smallest keyword which hits the cut. Similar to s-node, dummy cut will be explored (Line 10) since all subscriptions on the dummy cut survive the keyword filtering according to its definition. For each incoming message m, we retrieve all the matching subscriptions by calling the function IncomingMessageMatching(m, 1, *root*), where *root* is the root node of AP-Tree.

Example 6 Suppose 9 subscriptions in the running example (Fig. 3) are organized by AP-Tree as shown in Fig. 5c. For the incoming message m_1, we first access k_1-node with $\eta = 1$. According to Lines 5–10, the cut $c[w_2, w_3]$ on k_1-node is hit by both the *first* and *second* keyword w_2 and w_3 in m_1. Therefore, s_1-node will be explored with $\eta = 1 + 1 = 2$. Similarly, s_2-node is accessed with $\eta = 3 + 1 = 4$. Regarding s_1-node, we identify the grid cell stabbed by $m_1.loc$ (shaded cell on s_1-node) and reach the corresponding l-node, which contains $\{s_7\}$. We verify s_7 and put it into $\mathcal{R}$ because it satisfies both keyword and spatial constraints. The same procedure is applied to s_2-node. Since there is no l-node on the cell stabbed by m_1 (shaded cell on s_2-node), none of the child l-nodes of s_2-node will be accessed. Finally, we have $\mathcal{R} = \{s_7\}$. In this example, the total number of subscriptions verified is only 1.

Time complexity The dominant cost of Algorithm 1 is the AP-Tree traverse cost and verification cost. The traverse costs are $O(|m.\psi| \times \log(f))$ and $O(\log(f))$ for each k-node and s-node, respectively. The verification cost of a subscription s is $O(|m.\psi| + |s.\psi|)$ in the worst case, while the number of verifications heavily depends on the filtering capability of AP-Tree.

Algorithm correctness Since each subscription will be validated at Line 2, it is immediate that all subscriptions in $\mathcal{R}$ are valid. As a subscription may be assigned to disjoint grid cells at each s-node and the union of these cells contains the

Algorithm 1: IncomingMessageMatching(m, η, N)

```
Input   : $m$ : incoming message
          $\eta$ : the start matching position w.r.t. $m.\psi$
          $N$ : node accessed currently
Output  : $\mathcal{R}$ : set of all the matching subscriptions
1  if $N$ is a l-node then
2      Verify subscriptions stored in s-list and insert the matching
       ones to $\mathcal{R}$;
3      return
4  if $N$ is a k-node then
5      for $\eta \le i \le |m.\psi|$ do
6          Find the corresponding $cut$ based on $w_i$ in $m.\psi$;
7          if $cut$ has not been visited then
8              IncomingMessageMatching($m, i+1, cut$);
9      if $dummy\_cut$ exists then
10         IncomingMessageMatching($m, \eta, dummy\_cut$);
11 else
12     Find the $cell$ which covers $m.loc$ using grid;
13     IncomingMessageMatching($m, \eta, cell$);
14     if $dummy\_cell$ exists then
15         IncomingMessageMatching($m, \eta, dummy\_cell$);
```

subscription range, each matching subscription s must be assigned to a l-node whose ancestor s-nodes are stabbed by the message location. Let $B_1, B_2, \dots, B_j$ denote the cuts or cells along the path from $root$ of AP-Tree to this l-node. It is immediate that cell B_1 will be visited if the root is a s-node. Similarly, the cut B_1 will be visited if the root is a k-node since there must exist one message keyword which is equal to the first subscription keyword of s. It is easy to see that B_i will be visited sequentially for $1 < i \le j$, and the correctness of Algorithm 1 follows.

3.3 AP-Tree construction and maintenance

We first propose a cost model in Sect. 3.3.1 to quantitatively analyze the goodness of keyword and spatial partitions. Then efficient keyword and spatial partition approaches are devised to minimize the matching cost in Sects. 3.3.2 and 3.3.3, respectively. Section 3.3.4 presents the AP-Tree construction algorithm which adaptively selects keyword and spatial partition methods to construct AP-Tree in a top-down manner. Section 3.3.5 develops dynamic maintenance approach which makes AP-Tree self-adjustable to the change of subscription workload.

3.3.1 Cost model

Given a set $\mathcal{S}$ of subscriptions, AP-Tree is constructed in a top-down manner. Thus, we need to evaluate the goodness of a keyword or spatial partition such that the AP-Tree is adaptive to subscription workload. In this section, we propose a cost model to quantitatively measure the matching cost for two partition methods. Given a node N and a set S of subscriptions assigned to N, without further partition the matching cost contributed by N is $|S|$, assuming the average subscription verification cost is a unit time. Clearly, we can partition $|S|$ subscriptions into a set $\mathcal{P}$ of f cuts or cells by *keyword partition* or *spatial partition* to reduce the matching cost.

Let B denote a cut or cell of the partition, we use $\mathbf{w}(B)$ to record its *weight* which is the number of subscriptions associated with B. By $\mathbf{p}(B)$, we mean the *hit probability* of B, i.e., the probability that B is explored during the message matching. The expected matching cost regarding partition $\mathcal{P}$, denoted by $C(\mathcal{P})$, is:

$$C(\mathcal{P}) = \sum_{i=1}^{f} \mathbf{w}(B_i) \cdot \mathbf{p}(B_i) \tag{1}$$

Given a partition $\mathcal{P}$ and a set of subscriptions S on the node, the calculation of $\mathbf{w}(B)$ is immediate for each B. We may derive the hit probability $\mathbf{p}(B)$ based on some distribution assumptions or message workload. For analysis simplicity, we assume that $\mathbf{p}(B) = \sum_{w \in B} \mathbf{p}(w)$ for k-node, where $\mathbf{p}(w)$ is the hit probability of the keyword w. In case a set M of the messages is available, it is trivial to derive hit probability of each individual keyword. Otherwise, we assume the subscription keyword with high frequency among S has better chance to appear in message keywords; that is, we use subscription workload to simulate message workload. Specifically, we set $\mathbf{p}(w) = \frac{freq(w)}{\sum_{w \in \mathcal{P}} freq(w)}$ where $freq(w)$ is the frequency of keyword w among all subscriptions in S. Regarding *spatial partition*, we may simply assume the uniform distribution of the message location, and hence $\mathbf{p}(B) = \frac{Area(B)}{Area(N)}$ where $Area(B)$ is the area of the cell B and $Area(N)$ is the area of the node N. The hit probability calculation of each cell is immediate when message workload is available.

3.3.2 Keyword partition

Without loss of generality, we assume the lth keywords of the subscriptions in S correspond to a set of *ordered* keywords $V = \{w_1, w_2, \dots, w_{|V|}\}$. On each k-node, subscriptions are partitioned into f *ordered cuts* based on their lth keywords, and we aim to find an optimal keyword partition, denoted by $\mathcal{P}_k^*$, such that the matching cost is minimized. We first present a dynamic programming approach to achieve the optimal partition, followed by a simple optimal solution for a special case. Then we develop an efficient heuristic approach.

(1) Optimal partition

Dynamic programming algorithm By $\mathcal{P}_k(i, j, c)$, we mean a keyword partition regarding keywords between w_i and w_j (both inclusive) with c cuts. The *optimal partition* is denoted by $\mathcal{P}_k^*(i, j, c)$. Since keywords are ordered, we can come up with $\mathcal{P}_k^*(i, j, c)$ by exhausting all possible locations of the first cut as follows.

$$C(\mathcal{P}_k^*(i, j, c)) = \min_{i \le n \le j-c+1} (C(\mathcal{P}_k^*(i, n, 1)) + C(\mathcal{P}_k^*(n+1, j, c-1))) \quad (2)$$

Let $\mathcal{P}_k^*(i, n, 1)$ represent the optimal partition which consists of one cut $c[w_i, w_n]$, we have

$$C\big(\mathcal{P}_k^*(i, n, 1)\big) = \left(\sum_{j=i}^{n} \mathbf{w}(w_j)\right) \cdot \left(\sum_{j=i}^{n} \mathbf{p}(w_j)\right) \quad (3)$$

where $\mathbf{w}(w_j)$ denotes the number of subscriptions whose lth keyword equals w_j.

Algorithm 2 illustrates our dynamic programming method for optimal keyword partition. In particular, Lines 1–2 compute the cost for each partition with single cut. Then Lines 3–5 iteratively compute the optimal partitions with c cuts ($2 \le c \le f-1$). Finally, the optimal keyword partition $\mathcal{P}_k^*$ corresponds to $\mathcal{P}_k^*(1, |V|, f)$. The time complexity of Algorithm 2 is $O(f \cdot |V|^2)$.

Algorithm 2: OptimalKeywordPartition(V, f)

```
Input  : $V$ : keyword set to be partitioned
         $f$ : number of cuts
Output : $\mathcal{P}_k^*$ : optimal keyword partition
1 for $1 \le i \le j \le |V|$ do
2     Compute $C\big(\mathcal{P}_k^*(i, j, 1)\big)$ based on Equation 3 ;
3 for $2 \le c \le f-1$ do
4     for $1 \le i \le |V|+1-c$ do
5         Compute $C\big(\mathcal{P}_k^*(i, |V|, c)\big)$ based on Equation 2 ;
6 Compute $C\big(\mathcal{P}_k^*(1, |V|, f)\big)$ based on Equation 2 ;
7 return $\mathcal{P}_k^*(1, |V|, f)$
```

Optimal solution for special case We say the subscription workload and message workload have similar distribution if and only if $\frac{\mathbf{p}(w_i)}{\mathbf{w}(w_i)} = \lambda$ for any $1 \le i \le |V|$. In this special case, we come up with a simple optimal solution with time $O(|V|)$ if each cut has the same weight. In particular, the cost model in Eq. 1 now turns to

$$C(\mathcal{P}) = \lambda \sum_{i=1}^{f} \mathbf{w}(B_i)^2 \quad (4)$$

According to Cauchy–Schwarz inequality, we have $(\sum_{i=1}^{f} \mathbf{w}(B_i)^2)(\sum_{i=1}^{f} 1^2) \ge (\sum_{i=1}^{f} \mathbf{w}(B_i) \cdot 1)^2$. Therefore, $C(\mathcal{P})$ can achieve the optimal solution if $\mathbf{w}(B_i) = \mathbf{w}(B_j)$ for $1 \le i, j \le f$. Note that as discussed in Sect. 3.3.1, we use subscription workload to simulate message workload when message workload is unavailable, and hence two distributions are similar.

Algorithm 3: HeuristicKeywordPartition(V, f)

```
Input  : $V$ : keyword set to be partitioned
         $f$ : number of cuts
Output : $\mathcal{P}_k$ : keyword partition
1 Find a partition $\mathcal{P}_k$ which evenly partitions $V$ by weight;
2 for $2 \le i \le f$ do
3     for each keyword $w$ between $l(c_{i-1})$ and $r(c_i)$ do
4         Compute $C(\mathcal{P}_k)$ suppose $c_{i-1}$ and $c_i$ are separated by $w$;
5         Update $c_{i-1}$ and $c_i$ in $\mathcal{P}_k$ using $w$ if a lower $C(\mathcal{P}_k)$ is
          achieved;
6 return $\mathcal{P}_k$
```

(2) Heuristic partition
Following the *local improvement heuristic* [38], we develop an efficient greedy partition algorithm, where details are illustrated in Algorithm 3. Line 1 first partitions V into f cuts with similar weights. Then Lines 2–5 iteratively improve keyword partition method by exhaustive search in a local area. In particular, let c_i denote the ith ordered cut, while $l(c_i)$ and $r(c_i)$ represent its left and right boundary keywords, respectively. For each cut c_i ($1 < i \le f$), we attempt to reduce the *local* cost (i.e., the cost of c_{i-1} and c_i) by exhausting all possible boundary (separate) keywords regarding two adjacent cuts c_{i-1} and c_i. The time cost of Algorithm 3 is $O(f \cdot |V|)$ in the worst case.

3.3.3 Spatial partition

Without loss of generality, we assume $f = x \times y$ and $\mathcal{P}_s$ represents a spatial partition of the node N which divides the region into $x \times y$ grid cells. We first show that it is an NP-hard problem to find optimal spatial partition. Then we resort to local improvement heuristic algorithm.

Theorem 1 *The problem of finding optimal spatial partition is* ***NP-hard****.*

Proof Our proof relies on the problem of generalized block distribution (GBD) [18] with $K = 1$, which is NP-complete. **GBD Instance** Given a $g \times g$ matrix $\mathbf{A}$, and each element is an integer; A partition which divides $\mathbf{A}$ into $x \times y$ contiguous blocks where $\mathbf{B}_{i,j}$ denotes the ijth block; A function ϕ, where $\phi(\mathbf{B}_{i,j})$ reports the number of nonzero elements in block $\mathbf{B}_{i,j}$. **Question** Is there a partition on $\mathbf{A}$ such that

$$\max_{1 \le i \le x, 1 \le j \le y} \phi(\mathbf{B}_{i,j}) \le 1 \quad (5)$$

Figure 6a shows an example of GBD problem where each block contains at most one nonzero element under the given partition ($g = 4$, $x = y = 3$). Given an instance of GBD, we

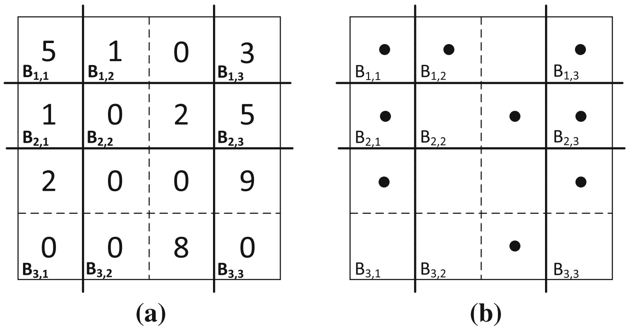


Fig. 6 Example of NP-complete. **a** Example of GBD problem. **b** Example of our problem

reduce it to a special case of decision version of our spatial partition problem as follows. Suppose there are $g \times g$ unit cells in the region of node N, as shown in Fig. 6b, we put a subscription with extremely small range (thus being regarded as a point) at the center of an unit cell if the corresponding element in $\mathbf{A}$ is nonzero. A spatial partition of node N divides the space into $x \times y$ grid cells. $\mathbf{w}(B_{i,j})$ $(1 \leq i \leq x, 1 \leq j \leq y)$ is the number of subscriptions in the cell $B_{i,j}$ and $\mathbf{p}(B_{i,j}) = \frac{\mathbf{w}(B_{i,j})}{|S|}$ where $|S|$ is the number of subscriptions generated. A special case of decision version of our problem is that whether there is a spatial partition $\mathcal{P}_s$ on the node N such that

$$C(\mathcal{P}_s) = \sum_{i=1}^{x}\sum_{j=1}^{y} \mathbf{w}(B_{i,j}) \cdot \mathbf{p}(B_{i,j}) \leq 1 \tag{6}$$

Since $\mathbf{p}(B_{i,j}) = \frac{\mathbf{w}(B_{i,j})}{|S|}$, we have $C(\mathcal{P}_s) = \frac{1}{|S|}\sum_{i=1}^{x}\sum_{j=1}^{y} \mathbf{w}(B_{i,j})^2$. Given the fact that $\sum_{i=1}^{x}\sum_{j=1}^{y} \mathbf{w}(B_{i,j}) = |S|$, a partition $\mathcal{P}_s$ with $C(\mathcal{P}_s) \leq 1$ implies that $\mathbf{w}(B_{i,j}) \leq 1$ for any cell $B_{i,j}$, i.e., there is at most one subscription in each cell. Note that if there exists one delimiter line of the spatial partition which lies across unit cells, we can simply shift it to its nearest boundary line without changing the partition cost. Thus, as illustrated in Fig. 6, $\mathcal{P}_s$ immediately leads to a solution of GBD problem where there is at most one nonzero element in each block, and vice versa. Thus, our problem is NP-hard.

Due to the NP-hardness of the problem, we resort to a *local improvement* heuristic algorithm in which the space is partitioned along each dimension independently. We first partition the space into x cells along the first dimension such that the centers of the subscriptions are evenly distributed. With similar rationale to Algorithm 3, we iteratively improve the partition cost. Since the possible number of boundary points along each dimension is bounded by $2 \cdot |S|$, the time complexity is $O(x \cdot |S|)$ in the worst case. Similarly, the space is partitioned into y cells along another dimension. In this way, we divide the region of N into f grid cells with time complexity $O(\sqrt{f} \cdot |S|)$.

3.3.4 Index construction

Algorithm 4 presents the procedure of AP-Tree construction, which recursively divides subscriptions through keyword and spatial partitions. Given a set S of subscriptions passed from parent node, the current node N may be set to l-node, k-node or s-node. Specifically, two flags, kP and sP, are used to indicate if subscriptions in S can be further partitioned by keyword and space, respectively. Line 2 keeps all subscriptions in the s-list of a l-node if the number of subscriptions does not exceed a given threshold θ_p (i.e., $|S| < \theta_p$) or subscriptions cannot be split further by keyword or spatial partitions (i.e., kP is $false$ and sP is $false$). If keyword partition is allowed (i.e., kP is $true$), Line 6 explores keyword partition with offset l, and the cost is recorded by C_k. Recall that offset l indicates that the lth keywords from subscriptions in S are employed for keyword partition. By C_s, we record the cost of spatial partition at Line 8 if sP is $true$. Then we can decide the current node N to be constructed from keyword partition (Line 10) or spatial partition (Line 18) based on C_k and C_s. The subscriptions in S are pushed to related child nodes (i.e., cuts and cells) for further processing (Line 16 and Line 24), in which the partition offset is increased by one if keyword partition is adopted.

In addition to regular cuts (cells), we also maintain dummy cut (cell) for k-node (s-node). In particular, we maintain a dummy cut for a k-node such that subscriptions whose keywords have been exhausted (i.e., $|s.\psi| < l$) are pushed to the dummy cut with kP set to $false$ (Lines 11–13). Similarly, Lines 19–21 push all subscriptions with ranges containing the node N to the dummy cell for further potential keyword partition, where the flag sP is set to $false$. Finally, the AP-Tree can be constructed by the function IndexConstruction$(root, \mathcal{S}, 1, true, true)$.

3.3.5 Index maintenance

In practice, we may need to dynamically maintain AP-Tree due to registration of new subscriptions and unregistration of existing subscriptions. A simple strategy is that we put a new subscription into its corresponding l-node based on its ordered keywords and search range, and a l-node is partitioned when its number of subscriptions exceeds the threshold θ_p. Similarly, we remove a subscription from its corresponding l-node if it is unregistered and a k-node or s-node degrades to a l-node if the number of its descendant subscriptions is less than θ_p. This approach is efficient and works well if the underlying subscription workload remains stable. On the downside, the partitions of the existing nodes cannot be adjusted to the change of subscription workload,

Algorithm 4: IndexConstruction(N, S, l, kP, sP)

```
Input  : $N$ : current node, $S$ : a set of subscriptions
         $l$ : keyword partition offset to be used in $N$
         $kP$ and $sP$ : flags for keyword and spatial partitions
Output : AP-Tree
1  if ($kP$ is $false$ and $sP$ is $false$) or $|S| < \theta_p$ then
2      Construct $N$ as a l-node, and add $S$ to the s-list ;
3      return
4  $C_k := +\infty$; $C_s := +\infty$;
5  if $kP$ is $true$ then          /* Try keyword partition */
6      $C_k \leftarrow$ keyword partition on $S$ with offset $l$;
7  if $sP$ is $true$ then          /* Try spatial partition */
8      $C_s \leftarrow$ spatial partition on $S$;
9  if keyword partition is chosen (i.e., $C_k < C_s$) then
10     Construct $N$ as a k-node with node offset $N_l := l$;
11     $S' \leftarrow$ subscriptions $\{s\}$ in $S$ with $|s.\psi| < l$;
12     $B' \leftarrow$ dummy cut of $N$;
13     IndexConstruction( $B', S', l+1, kP := false, sP$);
14     for each child node (i.e., cut) $B$ of node $N$ do
15         $S_B \leftarrow$ subscriptions in $S - S'$ which hit the cut $B$;
16         IndexConstruction($B, S_B, l+1, kP, sP$);
17 else
18     Construct $N$ as a s-node ;
19     $S' \leftarrow$ subscriptions in $S$ which contain $N_r$;
20     $B' \leftarrow$ dummy cell of $N$;
21     IndexConstruction($B', S', l, kP, sP := false$);
22     for each child node (i.e., cell) $B$ of node $N$ do
23         $S_B \leftarrow$ subscriptions in $S - S'$ which overlap or contain $B$;
24         IndexConstruction( $B, S_B, l, kP, sP$);
```

and hence the performance may be deteriorated. To alleviate this issue, we adopt the well-known *KL-Divergence* [26] to detect the changes of underlying subscription workload for nodes with a particular amount of subscriptions. Specifically, let $\mathbf{w}_{old}(B_i)$ denote the weight of the cut/cell B_i when the node is constructed, while $\mathbf{w}(B_i)$ is calculated for all current subscriptions. Let $D_{KL}(\mathbf{w}_{old}|\mathbf{w})$ denote *KL-Divergence* of the subscription workload, and an AP-Tree node will be reconstructed if $D_{KL}(\mathbf{w}_{old}|\mathbf{w})$ exceeds a given threshold θ_{KL}. We remark that calculation of *KL-Divergence* value is almost cost free because they can be easily updated when the node is visited during the subscription updates. Moreover, only descendant subscriptions of the node are involved in the reconstruction. In this way, our empirical study shows that AP-Tree is self-adjustable to the workload changes with a decent maintenance overhead.

3.4 Discussions

General boolean subscriptions In the above discussion, we only consider *ALL* matching semantic, where all the keywords of a subscription must be contained by a message. It is desirable to support more general boolean expressions, such as $w_1 \wedge (w_2 \vee w_3) \wedge w_4$, which has more powerful expressiveness and is well studied in traditional Pub/Sub systems (e.g., [35,41,48]). Our indexing structure can be easily extended to support general boolean expressions. Specifically, it is known that any boolean expression can be rewritten into *Disjunctive Normal Form (DNF)* [15], which is a disjunction of conjunctive clauses. Thus, we only consider the processing of DNF in the following. For DNF subscription, we can regard each conjunctive clause as a sub-subscription and decompose the DNF subscription into multiple sub-subscriptions, each with the same search range as the original one. A message will be delivered to the original subscription as long as any of its sub-subscriptions has been matched.

4 Moving Publish/Subscribe

In this section, we extend our AP-Tree framework to support moving subscriptions efficiently. Section 4.1 proposes preliminaries and the formal problem definition. The new indexing structure is investigated in detail in Sect. 4.2. A modified cost model and the issues related to index construction and maintenance are discussed in Sect. 4.3.

4.1 Preliminaries

In the section, we give some preliminaries and the formal definition of moving subscription problem.

Following the previous work on moving query processing [29,31,43], we employ a *centralized* framework, where the entire processing cycle takes place on the server side, while the client side can only send location update and receive results due to the limitation of local computational capability.

In a moving Pub/Sub system, a subscriber can register her interest as a moving spatial-keyword subscription, which is defined as $s = (\psi, r)$. The subscriber can report her location update at each timestamp. A spatial-textual message is defined as $m = (\psi, loc, t_c, t_e)$, where t_c and t_e are the creation and expiration time of m. Note that, for stationary subscriptions, each message is executed as a one-time snapshot matching, while for moving subscriptions, each message needs to be indexed for continuous re-evaluation.

Problem statement We consider a large number of moving spatial-keyword subscriptions $\mathcal{S}$ registered by subscribers and a stream of incoming messages $\mathcal{M}$, each having a life cycle. The subscribers can move randomly at any time, while the messages are coming in a stream fashion and will be expired after their life cycles. The moving Pub/Sub supports the following two operations.

- **Incoming message matching** For each new incoming message, it is delivered to all the *relevant* subscribers registered in the system instantly. This function is essentially the same as the problem defined in Sect. 3.1.

- **Moving subscription processing** For each moving subscriber, all the *relevant active* messages are reported to her continuously; that is, when subscriber issues a location update to the server, the server re-evaluates the *relevant* messages for the subscriber in real time.

To support the above two operations, we extend the original AP-Tree structure into a new structure, namely AP^+-Tree, by making some modifications to the original cost model. Also, to support moving subscription processing, we attach the active messages into AP^+-Tree directly, without the burden to build a new indexing structure. Other operations such as subscription insertion and deletion, message insertion and deletion can also be supported efficiently.

For the ease of explanation, we call the indexes for subscriptions and messages as SubIndex and MsgIndex, respectively, in the following of the paper.

4.2 AP^+-Tree framework

In this section, we first introduce the motivations behind our new indexing structure AP^+-Tree (Sect. 4.2.1), followed by an overview of the indexing structure (Sect. 4.2.2) and the moving Pub/Sub (Sect. 4.2.3). The details of *Incoming Message Matching* (Sect. 4.2.4) and *Moving Subscription Processing* (Sect. 4.2.5) are investigated, respectively at last.

4.2.1 Motivation

Motivation 1 *We need to modify the previous cost model to accommodate the movement of subscriptions.* The main difference of moving subscriptions compared to stationary ones is that a moving subscriber has to update her location continuously, and thus the reconstruction of the indexing structure may be triggered due to the adjustment of underlying spatial distribution. Therefore, it is very intuitive to modify the previous cost model such that the differences incurred by the movement of subscriptions can be considered. To achieve this, we add some extra costs to s-node as a penalty; we do not touch k-node because only s-node may be influenced by moving subscribers. This penalty should reflect the motion patterns of subscriptions. For example, if there are too many moving subscribers or the subscribers in a region issue location update very frequently, the penalty should be large. In this way, our indexing structure can automatically adjust itself to give less priority to the s-node which may trigger large moving penalty, thus being adaptive to not only the keyword and spatial distributions of subscriptions but also the movement of subscriptions.

Motivation 2 *Both subscriptions and messages should be indexed efficiently for real-time response.* Given the massive volume of subscriptions and messages, it is critical to design efficient indexing approaches to support real-time processing. An immediate solution is to organize subscriptions and messages separately. For example, we can organize subscriptions using AP-Tree structure and organize messages using either *keyword-prioritized* or *spatial-prioritized* indexing structure. The main drawback of the naive solution is that the organization of messages cannot take advantages of the organization of subscriptions, i.e., our AP-Tree structure, which has better filtering capability. On the other hand, we observe that, when we match incoming messages, each message has been distributed to its *relevant* subscriptions *only*, thus leading to a partition for messages implicitly. Based on this observation, we partition the messages following the same structure as subscriptions, and store each message only to the nodes where there exist *relevant* subscriptions. In this way, we group *relevant* subscriptions and messages together using a unified framework, avoiding the burden to build another index for messages. Experimental results verify the great efficiency and effectiveness of this strategy.

4.2.2 Index overview

Our AP^+-Tree is essentially a natural extension of AP-Tree structure. Specifically, there are two main differences. Firstly, we modify the original cost model to accommodate the movement of subscribers, considering the penalty introduced by moving subscriptions (Sect. 4.3.1). Secondly, as each message has a life window, we introduce a new list type, namely m-list, for each l-node to store the active messages which have visited this l-node during message matching procedure. A m-list is essentially a list of messages which are *relevant* to the subscriptions stored in the l-node. Figure 7 shows a framework of AP^+-Tree. As can be shown clearly, a l-node stores both s-list and m-list for subscriptions and messages, respectively.

4.2.3 System overview

In this section, we introduce the procedures to process moving spatial-keyword subscriptions. All the *relevant* messages are delivered to corresponding subscribers in an incremental manner to avoid continuous re-evaluation. Following the

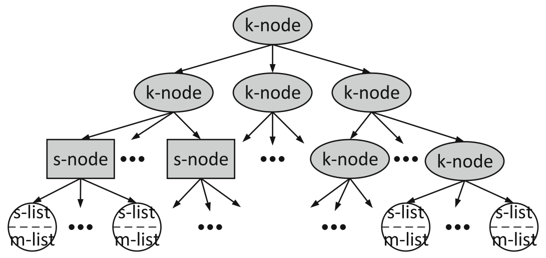


Fig. 7 Framework of AP^+-Tree

Algorithm 5: MovingPub/Sub

```
1 for each event tuple $event$ coming within timestamp $t$ do
2     if $event$ is from an incoming message $m$ then
3         $\mathcal{R}$ = IncomingMessageMatching($m, 1, root$);
4         for each subscription $s$ in $\mathcal{R}$ do
5             Deliver update $(s, +m)$ to corresponding subscriber;
6         InsertMessage($m$);
7     if $event$ is from a moving subscriber $s$ with new range $r_n$ then
8         MovingSubscriptionProcessing($s, s.r, r_n$);
9 LazyMessagesDeletion;
```

efficient incremental technique proposed in [29], we distinguish two types of updates: *positive* and *negative*. A positive update in the form of $(s, +m)$ indicates that the message m should be added to the result list of subscription s, while a negative update in the form of $(s, -m)$ suggests that m should be removed from the result list of s. Note that a positive update can cancel out previous negative update and vice versa. We define an *event* as either the arrival of a new message or the location update from a moving subscriber. The details of the system procedures are shown in Algorithm 5. For each event tuple arriving within current timestamp, if it is from an incoming message, we simply call function IncomingMessageMatching to retrieve all the subscriptions matching this message, and deliver the result updates to corresponding subscribers (Lines 3–5). We also need to insert this message into AP$^+$-Tree by calling function InsertMessage. If the event tuple is from a moving subscriber, we need to update the result list of this subscriber by calling function MovingSubscriptionProcessing (Line 8). This function works using incremental evaluation strategy, thus only retrieving the new results (i.e., positive updates) and deleting expired results (i.e., negative updates). Finally, we employ a *lazy* deletion strategy for the expired messages. Function LazyMessagesDeletion (Line 9), which will be triggered only when the memory is exhausted, or a predefined time interval for scheduled deletion has reached, deletes all the expired messages from AP$^+$-Tree. Note that we do not include the insertion of new subscriptions or expiration of old subscriptions in Algorithm 5 for simplicity, because they can be easily modeled as moving subscriptions. For example, the insertion of a new subscription can be regarded as moving this subscription from a far-away position to a nearby location. In the following sections, we introduce the above functions in detail.

4.2.4 Incoming message matching

As our new index AP$^+$-Tree is a natural extension of original AP-Tree structure with modified cost model and supplementary m-list in each l-node, we can reuse the previous message matching algorithm described in Algorithm 1 immediately. As to the insertion of message (Line 6 in Algorithm 5), instead of inserting a new message into AP$^+$-Tree after matching procedure separately, we insert it as a by-product of matching procedure, with little extra overhead. Specifically, for each l-node encountered (Lines 1–3 in Algorithm 1), we add a statement to insert the message into the corresponding m-list in l-node (Between Line 2 and Line 3 in Algorithm 1). This statement guarantees that a message will be stored in all the l-nodes where there exist *relevant* subscriptions. The correctness of this strategy will be proved in Sect. 4.2.5. We remark that, in order to avoid assigning too many duplicates of messages, we set a limit for the maximum number of duplicates. A message will not be further pushed to lower levels if this limit has been reached. The corresponding modification for moving subscription processing is to check not only the messages in l-nodes, but also the messages stored in the ancestor nodes.

4.2.5 Moving subscription processing

In a moving Pub/Sub system, each moving subscriber may report a location update to the centralized server in each timestamp. The system should be able to handle the update efficiently and re-evaluate the matching results in real time. In this section, we focus on efficiently re-evaluating *relevant* results in an incremental manner; that is, we only report positive/negative updates of previous results.

Algorithm 6 describes the details of moving subscription processing. In order to facilitate the moving subscription processing, for each subscription s, we keep track of a list, $node_{list}(s)$, of l-nodes where the subscription is stored. When a subscriber s issues a location update, MovingSubscriptionProcessing($s, s.r, r_n$) will be triggered, where $s.r$ is the old subscription range and r_n is the new subscription range. An example is shown in Fig. 8. In this example, we assume a subscription s moves from top-left to bottom-right. Each cell in this figure corresponds the region of a l-node, and s is indexed in all the cells which have overlaps with it. When a location update occurs, we process the l-node stored in $node_{list}(s)$ one by one.[8] For each l-node N (Line 4), we distinguish four cases to avoid repeated computations.

- **Case 1** The region of N is *fully* covered by both r and r_n (Lines 5–6). In this case, we do nothing to node N because the results remain the same for both old r and new r_n. This corresponds to the two darkest cells in Fig. 8.

[8] The l-nodes in current $node_{list}(s)$ are those covered by $s.r$, i.e., case 1, case 2 and case 3 in Fig. 8, while case 4 indicates the new l-nodes which are only covered by r_n.

Algorithm 6: MovingSubscriptionProcessing

```
Input  : $s$ : the moving subscription
         $r$ : the old range of $s$
         $r_n$ : the new range of $s$
Output : $upd_{list}(s)$ : a list of result updates for $s$
1  Initialize $upd_{list}(s) := \emptyset$;
2  Initialize $r' := \emptyset$;
3  Initialize $tmp_{list}(s) := node_{list}(s)$;
4  for each l-node $N$ in $tmp_{list}(s)$ do
5      if $(r \cap N_r) = (r_n \cap N_r)$ then                /* Case 1 */
6          Continue;
7      if $(r \cap N_r) \cap (r_n \cap N_r) = \emptyset$ then      /* Case 2 */
8          DeleteSubscription(s, $r \cap N_r$, $N$);
9          Continue;
10     if $(r \cap N_r) \cap (r_n \cap N_r) \neq \emptyset$ then      /* Case 3 */
11         DeleteSubscription(s, $(r \cap N_r) - (r_n \cap N_r)$, $N$);
12         InsertSubscription(s, $(r_n \cap N_r) - (r \cap N_r)$, $N$);
13         $r' := r' \cup ((r_n \cap N_r) - (r \cap N_r))$;
14         Continue;
15 InsertSubscription(s, $r_n - r - r'$, $root$)      /* Case 4 */;
16 return $upd_{list}(s)$
```

- **Case 2** The region of N is only covered (*fully* or *partially*) by r while has no overlap with r_n (Lines 7–9). In this case, we simply call function DeleteSubscription to delete s from N as well as issue negative updates for the corresponding results. This corresponds to the cells which are filled with horizontal lines in Fig. 8.
- **Case 3** The region of N is covered (*fully* or *partially*) by both r and n_r, while *fully* covered by at most one region (r or n_r) (Lines 10–14). In this case, we need to first delete expired results and then insert new results. This corresponds to the cells which are filled with slashes in Fig. 8. Note that we can avoid the re-evaluation of the intersection region of r and r_n (e.g., the small rectangle covered by the dotted circle).
- **Case 4** The region of N is only covered (*fully* or *partially*) by r_n while has no overlap with r (Line 15). In this case, we simply call function InsertSubscription to insert s into N as well as issue positive updates for the corresponding results. This corresponds to the cells which are filled with cross lines in Fig. 8.

The function InsertSubscription is shown in Algorithm 7. This algorithm is straightforward, and we omit the detailed explanation. The general idea is to follow the existing k-node or s-node until finally reaching l-node, where the subscription s will be stored. In each encountered l-node, we need to verify the subscription s against the messages stored in m-list to issue positive updates for the matching messages. We remark that the second parameter r' is only a subset of $s.r$, indicating that we only want to insert "partial" of s into the index. Thus, the verification step in Line 4 should be based on r', rather than $s.r$. In order to insert the entire

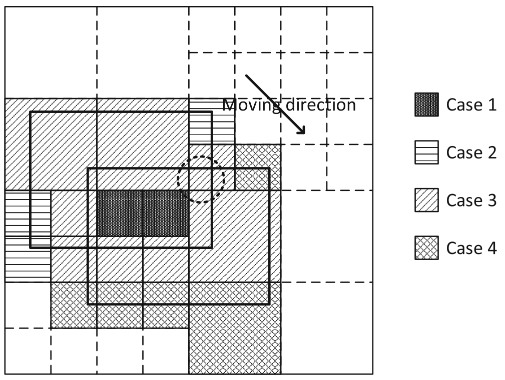


Fig. 8 Example of moving subscription processing

Algorithm 7: InsertSubscription(s, r', N)

```
Input  : $s$ : the moving subscription
         $r'$ : the partial search range of $s$ ($r' \subseteq s.r$)
         $N$ : current node in AP$^+$-Tree
1  if $N$ is a l-node && $r' \cap N_r \neq \emptyset$ then
2      Insert $s$ to the s-list of $N$;
3      for each message $m$ in m-list of $N$ do
4          if $m$ matches $s$ then      /* Matching here is
               conditioned on $r'$ */
5              Add result update $(s, +m)$ to $upd_{list}(s)$;
6      Add $N$ to $node_{list}(s)$;
7      return
8  if $N$ is a k-node then                      /* k-node */
9      Find the child node $B$ which contains $s$;
10     InsertSubscription($s$, $r'$, $B$) ;
11 else                                          /* s-node */
12     for each child $B$ which has overlap with $r'$ do
13         InsertSubscription($s$, $r'$, $B$);
14
```

s into our index, we can simply call InsertSubscription(s, $s.r$, $root$). We omit the *dummy* cuts or cells for simplicity. As the algorithm of subscription deletion is very similar to insertion, we also omit it.

The main advantages of our algorithm lie in two aspects. Firstly, incremental update strategy is applied to ensure we can avoid re-computing the regions where the results are still valid. Secondly, as we store messages using AP-Tree directly, the partition of messages can inherit the advantage of the partition of subscriptions, thus reducing the verification cost of *relevant* messages (Lines 3–5 of Algorithm 7).

Algorithm correctness We prove Algorithm 6 is correct. We first prove the incremental strategy is correct. This is obvious as our four cases cover all the possible situations (which is well illustrated in Fig. 8). We then prove that the results in $upd_{list}(s)$ is correct. Since all the messages will be checked at Line 4 in Algorithm 7, it is immediate that all the messages in $upd_{list}(s)$ are valid. According to the correctness of Algorithm 1 and the message insertion strategy proposed in Sect. 4.2.4, it follows that all the l-nodes containing subscriptions which are *relevant* to a message m will store m in its m-list. Thus, for each subscription s, all the *relevant* messages can be found from the m-list of l-nodes where s are stored. Thus, the correctness of the algorithm follows.

4.3 AP$^+$-Tree construction

In this section, we first propose the modified cost model to build AP$^+$-Tree (Sect. 4.3.1), and then discuss the index construction and maintenance (Sect. 4.3.2).

4.3.1 Cost model for moving subscriptions

In this section, we discuss the adaptation of previous cost model in Sect. 3.3.1 to support moving subscriptions.

The previous cost model only considers incoming message matching cost (C_{imm}). As to the moving case, one significant cost is location update cost of moving subscribers (C_{slu}). Location update cost can be regarded as a deletion followed by an insertion with new query range. Another important cost is index reconstruction cost (C_{irc}). When the subscribers move in space, the underlying spatial distribution of subscriptions may change continuously, which might violate the threshold θ_{KL} defined in Sect. 3.3.5 at some future timestamp. Thus, the reconstruction mechanism may be triggered, which leads to large system overhead. In the following sections, we will discuss the above three costs separately and quantitatively. To achieve this, we estimate the total cost occurred in *one timestamp*. We assume the moving pattern (e.g., velocity, direction, location update frequency) of all subscriptions remain unchanged during this timestamp. We denote the set of subscriptions assigned to node N as S.

Incoming message matching cost The cost of incoming message matching is almost the same as the cost defined in Eq. 1, except that we are now considering a timestamp. Thus, we need to estimate the number of incoming messages during one timestamp. Assume the incoming ratio of messages in each timestamp is r_m, and the verification cost of matching one subscription against one message is unit cost, the incoming message matching cost should be modified as:

$$C_{imm}(\mathcal{P}) = \sum_{i=1}^{f} (\mathbf{w}(B_i) \cdot \mathbf{p}(B_i) \cdot r_m) \tag{7}$$

where $\mathbf{p}(B_i) \cdot r_m$ is the number of incoming messages probing B_i within one timestamp. The value of r_m can be easily estimated from historical data. The time complexity to compute C_{imm} is $O(|S|)$.

Subscriber location update cost Subscriber location update cost refers to the cost incurred by the movement of subscribers. When a subscriber is moving, we need to continuously update her new location. It is intuitive that the location update cost incurred by a moving subscription s is the summation of the costs incurred on all the nodes which s resides in. Thus, for node N, we only consider the cost incurred on the intersection region between N and s, i.e., the region of s_1 in Fig. 9. We then regard each location update as a deletion followed by an insertion, and assume the insertion cost is the same as deletion cost. Thus, we can estimate the location update cost for a subscription which moves totally out of original node N (e.g., s_3 in Fig. 9) as:

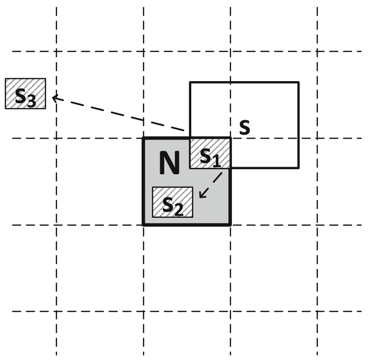


Fig. 9 Example of subscriptions with different velocities. We move s_1 to s_2 and s_3 with different velocities, respectively

$$C_{ins} = 2 \cdot l_{path} \cdot \log f \tag{8}$$

where l_{path} is the path length from root to N. $\log f$ is the cost to locate the corresponding cut/cell in each node on the path. Constant factor 2 indicates one deletion followed by one insertion operation.

However, the cost will be much smaller if a subscription simply moves inside the node (e.g., s_2 in Fig. 9). In order to reflect such difference, we add a *weight* ζ_s to each subscription s to model the effect of velocity. Intuitively, the faster s moves, the larger ζ_s will be. To compute ζ_s, we assume a future time interval T' from now on.[9] The moving velocity and direction of each subscription hold stable during T'. Our target is to estimate the number of N_r, denoted as $n_{g'}$, a subscription has passed during time interval T'. Then ζ_s is computed as the average number of N_r a subscription has passed in one timestamp, i.e., $n_{g'}/T'$. To better understand this idea, we duplicate the region of N in Fig. 9 to the infinite space using dotted lines. By moving s from s_1 to s_3, two N_r have been passed. Note that the selection of T' should guarantee that each subscription in S can pass at least one N_r during T'; otherwise, its factor ζ_s would be 0. Typically, the scale of T' does not affect ζ_s as long as T' is sufficiently large, because ζ_s is averaged upon T'. Finally, the cost of subscriber location update can be estimated as:

[9] T' here is essentially an amortized factor, such that the value of ζ_s will not depend on the length of one timestamp.

$$C_{slu}(\mathcal{P}) = \frac{1}{T_{upd}} \cdot \sum_{s \in S} \left(2 \cdot min(1, \zeta_s) \cdot \left(l_{path} \cdot \log f\right)\right) \quad (9)$$

where $min(1, \zeta_s)$ is to guarantee ζ_s is not larger than 1, because at most one deletion and insertion can occur in each timestamp. T_{upd} is the average time interval for a subscription to issue location update. It is intuitive from the formula that subscribers with fast moving velocity (large ζ_s) and frequent location update (small T_{upd}) will trigger large update cost. The time complexity of computing ζ_s for all subscriptions is $O(|S|)$. Thus, the total time complexity to compute C_{slu} is still $O(|S|)$.

Index reconstruction cost As the subscribers are moving in node N continuously, the underlying spatial distribution of node N may be varied, which might trigger the reconstruction operation defined in Sect. 3.3.5. The index reconstruction cost can be modeled as follows:

$$C_{irc}(\mathcal{P}) = \frac{C_{irc}(Tree(N))}{T_{\Delta}} \quad (10)$$

where T_{Δ} is the number of timestamps when the next reconstruction might be triggered from now. $Tree(N)$ is the sub-tree of node N and $C_{irc}(Tree(N))$ is the cost of reconstructing sub-tree of node N. In the following, we will estimate these two values, respectively. We first estimate $C_{irc}(Tree(N))$. Given the following facts, i.e., the number of subscriptions $|S|$ in node N, the fanout f of AP-Tree, and the partition termination threshold θ_p, it is easy to estimate the height of $Tree(N)$ as:

$$h = \log_f \left(\frac{|S|}{\theta_p}\right) \quad (11)$$

We ignore the effect of dummy node for simplicity.

For each s-node, the cost of reconstruction is $\sqrt{f} \cdot |S| + |S| \cdot \log f$, where $\sqrt{f} \cdot |S|$ is the cost to find best spatial partition $\mathcal{P}_s^*$ and $|S| \cdot \log f$ is the cost to assign each subscription to the corresponding cell. For each k-node, the cost of reconstruction is $f \cdot |V| + |S| \cdot \log f$, where $f \cdot |V|$ is the cost to find best keyword partition $\mathcal{P}_k^*$ and $|S| \cdot \log f$ is the cost to assign each subscription to the corresponding cut. Thus, the cost of reconstructing $Tree(N)$ is

$$(\sqrt{f} \cdot |S| + |S| \cdot \log f) + f \cdot \left(\sqrt{f} \cdot \frac{|S|}{f} + \frac{|S|}{f} \cdot \log f\right) + \cdots + f^h \cdot \left(\sqrt{f} \cdot \frac{|S|}{f^h} + \frac{|S|}{f^h} \cdot \log f\right) = |S| \cdot \left(\sqrt{f} + \log f\right) \cdot (h+1)$$

if we only use s-node.

Similarly, the cost of reconstructing $Tree(N)$ is $(f \cdot |V| + |S| \cdot \log f) \cdot (h+1)$ if we only use k-node.

Finally, the cost of reconstructing $Tree(N)$, if we use both s-node and k-node, is the summation of the above two costs[10]:

$$C_{irc}(Tree(N)) = \left(|S| \cdot \log f + |S| \cdot \sqrt{f} + f \cdot |V|\right) \cdot (h+1) \quad (12)$$

Next, we estimate the value of T_{Δ}. Based on the current velocity and direction for each subscription, as well as the assumption of uniform grid partition for s-node due to simplicity, it is easy to estimate the accurate location of each subscription at each timestamp t_i if we assume current timestamp is t_0. Algorithm 8 gives an algorithm to estimate T_{Δ}.

Algorithm 8: EstimateTimeIntervalForNextReconstruction

```
Input  : $S$ : set of subscriptions in node $N$
Output : $T_\Delta$ : number of timestamps for next reconstruction
1 $T_\Delta := 0$;
2 Initialize the spatial distribution $\mathbf{w}_{old}$ based on locations of
  subscriptions in $S$ at timestamp $t_0$;
3 for each next timestamp $t_i$ do
4     $T_\Delta := T_\Delta + 1$;
5     Update the spatial distribution $\mathbf{w}$ based on current
      subscription locations;
6     if $D_{KL}(\mathbf{w}_{old}|\mathbf{w}) > \theta_{KL}$ then
7         return $T_\Delta$
8     if $T_\Delta > T_{max}$ then
9         return $T_{max}$
```

Algorithm 8 works as follows. It first initializes the spatial distribution $\mathbf{w}_{old}$ of all the subscriptions in node N based on the initial locations (Line 2). Then, for each next timestamp t_i ($i \in \{1, 2, 3, \ldots\}$), we increment T_{Δ} by one (Line 4) and update $\mathbf{w}$ based on current new locations (Line 5). If the KL-Divergence between the old and new distributions is larger than a threshold θ_{KL} (Line 6), we return current T_{Δ} as answer because a reconstruction will be triggered according to the condition defined in Sect. 3.3.5. The time complexity of Algorithm 8 is $O(|S| + |S| \cdot \frac{T_{\Delta}}{T_{upd}})$, where $|S|$ is the cost for initialization (Line 2) and $|S| \cdot \frac{T_{\Delta}}{T_{upd}}$ is the number of subscription updates within T_{Δ}. Note that T_{max} in Line 9 is a upper bound for T_{Δ} and can be estimated according to reconstruction frequency of historical data.

Finally, we have:

$$C_{irc}(\mathcal{P}) = \left(|S| \cdot \log f + |S| \cdot \sqrt{f} + f \cdot |V|\right) \cdot \left(\log_f \left(\frac{|S|}{\theta_p}\right) + 1\right) \cdot \frac{1}{T_{\Delta}} \quad (13)$$

[10] Note that the subscriptions are assigned to cut or cell only once. Thus the coefficient of $|S| \cdot \log f$ is 1.

In Eq. 13, there are two important parameters which affect the index reconstruction cost: number of subscriptions $|S|$ and number of timestamps for next reconstruction T_Δ. Large $|S|$ and small T_Δ will incur high reconstruction cost. Thus, during AP-Tree construction, a node with large $|S|$ and small T_Δ may be preferred by k-node instead of s-node. This is also consistent with our intuition to penalize s-node. The time complexity to compute C_{irc} is the same as Algorithm 8.
Discussion Based on above three costs, now we can reach the costs for k-node and s-node, respectively:

$$C(\mathcal{P}_k) = C_{imm}(\mathcal{P}_k) \tag{14}$$
$$C(\mathcal{P}_s) = C_{imm}(\mathcal{P}_s) + C_{slu}(\mathcal{P}_s) + C_{irc}(\mathcal{P}_s) \tag{15}$$

As C_{slu} and C_{irc} are incurred by movement of subscriptions, which does not influence the cost of k-node, the cost of k-node consists of only the first cost C_{imm}, while the cost of s-node includes all the three costs. By means of adding two extra penalties to s-node, we can ensure that a s-node will not be preferred by our cost model if the subscribers in it are moving fast and frequently. From the perspective of indexing structure, our new cost model can actually push some undesirable s-nodes to the lower levels, such that it can be more adaptive to the moving subscriptions. Finally, we remark that all the three costs can be computed efficiently, with the complexity linear to $|S|$.

4.3.2 Index construction and maintenance

As we only extend the cost model of AP-Tree, it follows the same index construction algorithm (Algorithm 4) to construct AP$^+$-Tree. The algorithm to find optimal keyword partition is also the same as previous method. In terms of spatial partition, the computation of $C_{imm}(\mathcal{P}_s)$ can still follow the previous algorithm to minimize the matching cost. As we assume uniform grid partition for the estimation of $C_{slu}(\mathcal{P}_s)$ and $C_{irc}(\mathcal{P}_s)$, we can compute them directly and add them as a constant to the overall cost. As to the maintenance issue, we follow the same solution proposed in Sect. 3.3.5. The only difference is that, we also need to update the spatial distribution when a subscription reports a location update.

5 Experiments

In this section, we present results of a comprehensive performance study to evaluate the effectiveness and efficiency of our techniques proposed in this paper. All experiments are implemented in C++ and conducted on a PC with 3.4 GHz Intel Xeon 2 cores CPU and 32 GB memory running Red Hat Enterprise Linux. Following the typical setting of Pub/Sub systems (e.g., [28,48]), we assume indexes are fit in the main memory to support real-time response.

5.1 Experiments for stationary subscriptions

In this section, we evaluate the efficiency and effectiveness of AP-Tree for stationary subscriptions.

5.1.1 Experimental setup

To the best of our knowledge, IQ-Tree [6] and R^t-Tree [28] are only two existing work investigating the same problem as ours. Both work fall in the category of *spatial-prioritized* indexing structure. For comprehensive performance evaluation, we also investigate a *keyword-prioritized* indexing structure, namely RQ-Tree. In a nutshell, we implement and evaluate following algorithms.

- **R^t-Tree** Message matching algorithm based on R^{t++}-Tree proposed in [28], which achieves the best performance compared with R^t-Tree and R^{t+}-Tree. The source code is provided by the authors in [28].
- **IQ-Tree** Message matching algorithm based on IQ-Tree proposed in [6]. The cost model of subscription decomposition proposed in [6] is adopted to allocate subscriptions to Quadtree cells according to subscription and message workloads.[11]
- **RQ-Tree** The representative of *keyword-prioritized* indexing method which can be regarded as a variant of IQ-Tree. Particularly, RQ-Tree first employs ranked-key Inverted List [6,46] to partition subscriptions into the posting lists according to their *least frequent* keywords. Then for subscriptions on each posting list, we build a Quadtree for spatial filtering purpose where the cost model in [6] is also adopted.
- **AP-Tree** AP-Tree based message matching algorithm proposed in this paper. By default, the heuristic algorithms are employed for keyword and spatial partitions.

Datasets Four datasets are collected for experimental evaluations. **TWEETS** is a real-life dataset collected from Twitter [28], containing 12 million tweets with geo-locations from May 2012 to August 2012. *TWEETS* is the default dataset in the experiments. **GN** is obtained from the US Board on Geographic Names[12] in which each message is associated with a geo-location and a short text description. **CARS** and **AIS** obtain the geo-locations from Chorochronos Archive[13] and we randomly tag the locations with user-generated key-

[11] As we assume indexes are fit in the main memory, we use the number of verifications to evaluate the goodness of the subscription decomposition, instead of the number of I/Os.

[12] http://geonames.usgs.gov.

[13] http://www.chorochronos.org.

Table 2 Datasets statistics

Datasets	TWEETS	GN	CARS	AIS
Number of messages	12.7 M	2.2 M	2.2 M	5.7 M
Vocabulary size	1.7 M	208 K	81 K	81 K
Avg. number of keywords in messages	9	7	30	50

words from 20 Newsgroups.[14] The statistics of four datasets are summarized in Table 2.

Subscription workload We generate four subscription workloads based on the above four datasets. In each subscription workload, 5 M spatial-textual messages are randomly chosen from corresponding dataset. For each sampled message, we randomly pick j terms as subscription keywords and j is a random number between 1 and 5. The search range is set to a rectangle centered at the geo-location of the message, and the range size is uniformly chosen between 0.01 and 1 % of data space. Besides, we generate subscriptions with general boolean expressions as follows. For a set of keywords, we partition it into two subsets by randomly assigning a disjunction or conjunction between any two consecutive keywords. For each subset, a disjunctive or conjunctive clause is then generated randomly. For example, for the given keywords $\{w_1, w_2, w_3, w_4\}$, we may first partition it into two subsets, i.e., $\{w_1, w_2\}$ and $\{w_3, w_4\}$, by assigning a conjunction between w_2 and w_3. Then we assign a disjunction to form a clause $w_1 \vee w_2$ for left subset, while a conjunction to form a clause $w_3 \wedge w_4$ for right subset. Finally, we get a boolean expression of $(w_1 \vee w_2) \wedge w_3 \wedge w_4$.

Message workload We use first 5 % of the spatial-textual messages as the historical message workload when IQ-Tree, RQ-Tree and AP-Tree are constructed. The remaining messages are fed to the continuous subscriptions as streaming spatial-textual data.

The *average* message matching time is reported to evaluate the performance of the algorithms. We also evaluate the index construction and maintenance time as well as the index size. By default, keywords in vocabulary are sorted in decreasing order of their term frequencies over the subscription keywords. Important parameters of AP-Tree and alternative implementations are investigated in Sect. 5.1.2. Throughout the experiments, we set fanout f, partition threshold θ_p and *KL-Divergence* threshold θ_{KL} to 200, 40 and 0.001, respectively, unless otherwise specified.

[14] http://people.csail.mit.edu/jrennie/20Newsgroups.

5.1.2 Experimental tuning

Effect of f and θ_p. In the first set of experiments, we evaluate the impact of the fanout f and partition threshold θ_p in four datasets under default settings. Intuitively, a small f cannot fully utilize the keyword partition due to the small number of cuts on each k-node. On the other hand, a large f may result in poor adaptiveness of the AP-Tree. This is confirmed in Fig. 10a, where the average matching time is reported with f varying from 50 to 800. We set f to 200 for all datasets in the hereafter experiments. Figure 10b reports the average matching time as a function of θ_p which grows from 5 to 400. It is observed that θ_p does not noticeably affect performance when θ_p is smaller than 40. By default, θ_p is set to 40 for a better trade-off between index size and matching performance.

Comparison of AP-Tree variants. We compare the performance of several variants of AP-Tree as follows. **DP** employs dynamic programming approach to find optimal keyword partition, and **HR** uses the heuristic keyword partition. **KPriority** puts high priority to keyword partition on each node when AP-Tree is constructed, while spatial partition is prioritized in **SPriority**. Finally, **Trie-Qd** adopts the ordered keyword trie structure in [22] to organize subscriptions and then uses Quadtree to further partition subscriptions with the same keywords. Figure 11a, b report the average message matching cost and the index construction time of the algorithms, respectively, over four datasets where the default average subscription range size is set to 0.001 %. Following are two important observations.

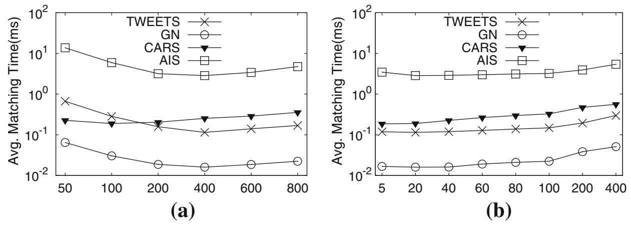


Fig. 10 Effect of varying f and θ_p. **a** Vary f **b** Vary θ_p

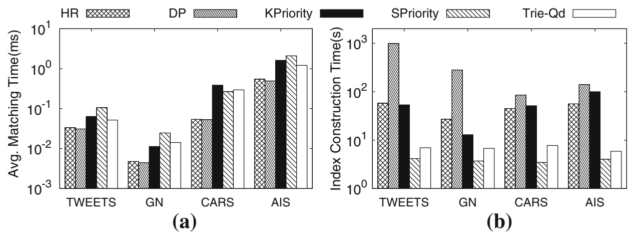


Fig. 11 Comparison among different AP-Tree variants. **a** Avg. matching time. **b** Index construction time

- Among all algorithms, *DP* achieves the best matching performance. *HR* has similar matching time with *DP* but beats *DP* by a huge margin w.r.t. index construction time.
- The poor matching performance of *KPriority* and *SPriority* implies that AP-Tree should be constructed in an adaptive way. Similarly, due to the lack of the adaptiveness and a large number of tree nodes, a straightforward combination of the ordered keyword trie [22] and Quadtree (*Trie-Qd*) cannot well support spatial-keyword subscriptions.

In hereafter experiments, *HR* is employed for performance evaluation of AP-Tree.

5.1.3 Performance evaluation

Evaluation on different datasets We evaluate the average message matching time, index construction time and index size of the algorithms against four datasets *TWEETS*, *GN*, *CARS* and *AIS*. As shown in Fig. 12a, AP-Tree significantly beats other algorithms in terms of message matching time. Particularly, AP-Tree is 30 times faster than the second best algorithm in *GN* because it is observed that the keyword and spatial distributions vary significantly among different regions in *GN*, and AP-Tree can take great advantage of its adaptiveness. It is worth noting that the *keyword-prioritized* method RQ-Tree has better performance than two *spatial-prioritized* methods (i.e., IQ-Tree and R^t-Tree) on *TWEETS*, *GN* and *AIS* datasets, but is defeated on *CARS* dataset by IQ-Tree. This implies that the effectiveness of the keyword and spatial filtering depends on the underlying subscription workload. Please note that IQ-Tree runs much faster than R^t-Tree, because IQ-Tree can benefit from its *space-oriented* partition strategy (i.e., Quadtree) rather than *object-oriented* partition method (i.e., R-Tree). As expected, Fig. 12b reports that R^t-Tree has the fastest index construction time because there is no cost model in [28] and the ranges of subscription are not decomposed. Figure 12c shows that the memory consumption of AP-Tree is comparable with other algorithms. In the following experiments, we exclude R^t-Tree from the performance evaluation because it is dominated by IQ-Tree.

Moreover, both algorithms belong to *spatial-prioritized* and hence exhibit similar trend in the experiments.

Effect of number of subscription keywords Figure 13 evaluates the performance of three algorithms against *TWEETS* and *GN* datasets where the number of subscription keywords varies from 1 to 5. Not surprisingly, the performance of three algorithms improves with the growth of the number of subscription keywords because the number of matching subscriptions is significantly reduced. When there is only one subscription keyword, AP-Tree only slightly outperforms RQ-Tree and IQ-Tree because it is difficult to distinguish subscriptions from keyword perspective. Nevertheless, the margin becomes significant when there are more than one subscription keyword.

Effect of subscription range size We evaluate the effect of subscription range size in Fig. 14 where the average matching time is reported as a function of the range size varying from 0.000001 to 10 % of the data space. As expected, the performance of three algorithms is sensitive to the range size because larger range size increases the number of matching subscriptions and hence leads to higher matching costs. It is noticed that RQ-Tree is ranked after IQ-Tree when

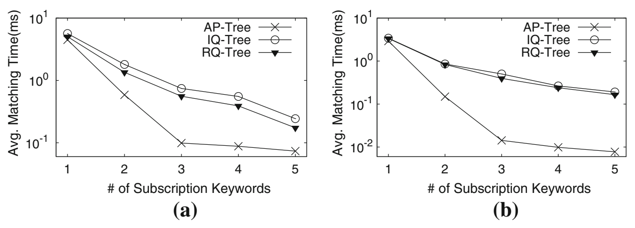


Fig. 13 Effect of number of subscription keywords. **a** TWEETS. **b** GN

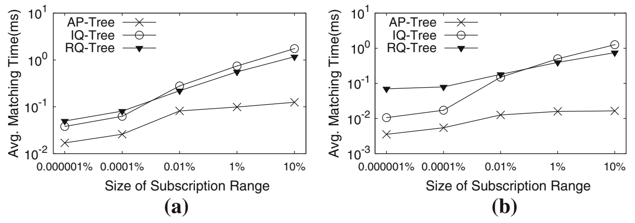


Fig. 14 Effect of subscription range size. **a** TWEETS. **b** GN

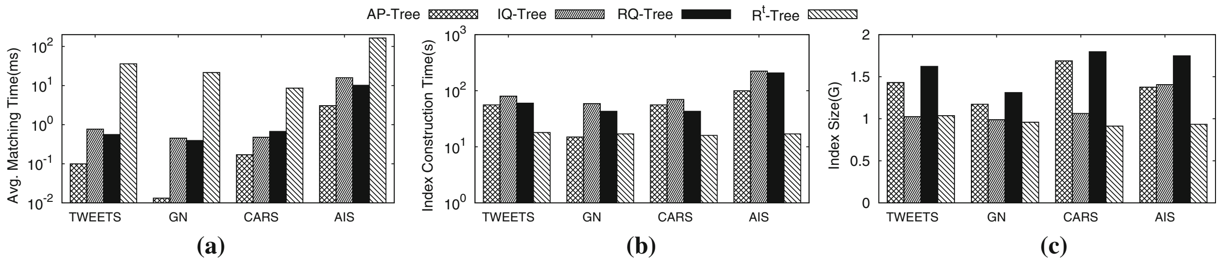


Fig. 12 Performance over various datasets. **a** Avg. matching time. **b** Index construction time. **c** Index size

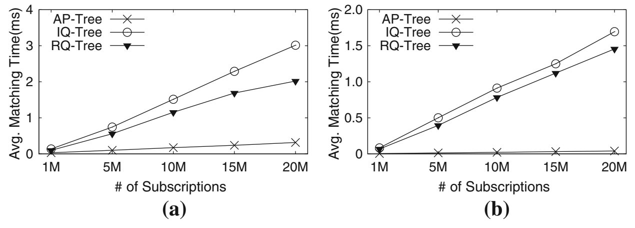


Fig. 15 Effect of number of subscriptions. **a** TWEETS. **b** GN

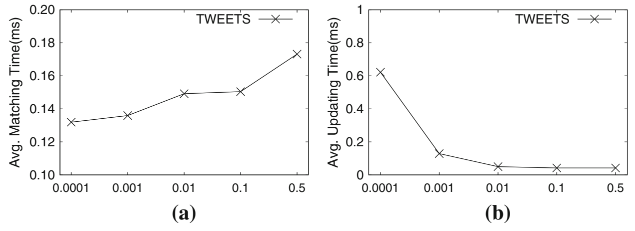


Fig. 16 Effect of θ_{KL}. **a** Vary θ_{KL}. **b** Vary θ_{KL}

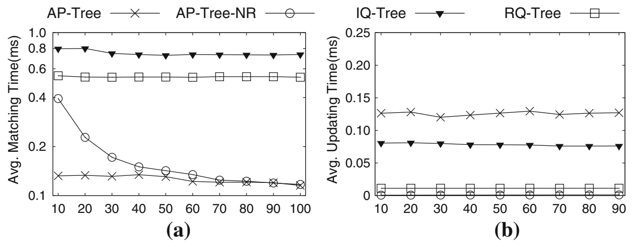


Fig. 17 Performance of maintenance. **a** Vary δ. **b** Vary δ

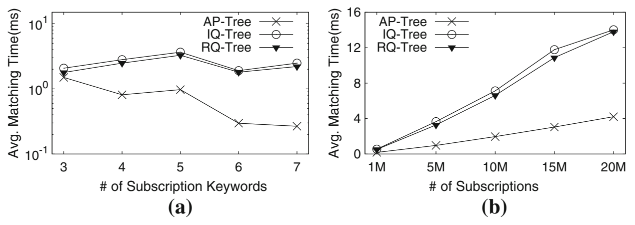


Fig. 18 Performance of general boolean subscriptions. **a** TWEETS. **b** TWEETS

the range size is very small, while RQ-Tree has better performance when the range size becomes large. This is quite intuitive because *spatial-prioritized* index is more attractive when the range size is very small. AP-Tree is the most stable algorithm and consistently beats RQ-Tree and IQ-Tree by a large margin. It is observed that more k-nodes appear on high levels of AP-Tree when the subscription range is large, which verifies the adaptiveness of AP-Tree structure.

Effect of number of subscriptions We turn to evaluate the effect of number of subscriptions in Fig. 15. We increase the number of subscriptions from 1 to 20 M. The result shows that AP-Tree is much more scalable to the number of subscriptions. For instance, it only takes 0.4 and 0.04 ms on average to match incoming messages on *TWEETS* and *GN* datasets when the number of subscriptions reaches 20 M.

Evaluate index maintenance We evaluate the costs of incremental maintenance of AP-Tree, IQ-Tree and RQ-Tree as well as their message matching performance. In particular, *TWEETS* dataset is deployed because the arrival order of the subscriptions can naturally follow the corresponding timestamps of the tweets. The first δ percentage of the subscriptions are used to construct the indexes, and then remaining subscriptions are incrementally inserted, where δ is set to 20 by default. Finally, we report the average message matching cost after all subscriptions arrive. We also record the average updating time for all subscriptions inserted.

In the experiments, a k-node or s-node of AP-Tree is reconstructed if it covers at least 0.1 % of the subscription population and its *KL-divergence* value exceeds θ_{KL}. It is quite intuitive that a small θ_{KL} value results in a better message matching time but higher AP-Tree maintenance overhead. Figure 16 evaluates the impact of threshold θ_{KL} which increases from 0.0001 to 0.5. In the following experiments, we set θ_{KL} to 0.001 since it achieves a good trade-off between matching cost and maintenance cost.

In this set of experiments, we also consider a variant of AP-Tree algorithm, namely AP-Tree-NR, which does not reconstruct the existing AP-Tree node. Figure 17 reports the average message matching time as well as the average delay of subscription insertions for four algorithms where the percentage of subscriptions used for initial AP-Tree construction (δ%) increases from 10 to 90 %. Figure 17a shows that the performance of AP-Tree-NR is not satisfactory when δ is small. This is because AP-Tree structure built on a small proportion of the subscription set does not well suit to the change of subscription workload. On the contrary, the performance of AP-Tree is rather stable and consistently beats IQ-Tree and RQ-Tree by a large margin since AP-Tree can adjust the tree structure to the change of subscription workload by node reconstructions. The average maintenance cost of four algorithms is reported in Fig. 17b. As expected, AP-Tree-NR has the best performance since there is no node reconstructions, while AP-Tree has the largest index maintenance overhead. Nevertheless, AP-Tree can process a subscription in around 0.12 ms on average which is still quite efficient in practice.

Evaluate general boolean subscriptions In this set of experiments, we evaluate the performance of processing general boolean subscriptions against *TWEETS* dataset in Fig. 18. We extend IQ-Tree and RQ-Tree to support general boolean subscriptions following the similar strategy as our AP-Tree. In Fig. 18a, we vary the number of subscription keywords from 3 to 7. It is noticed that all the algorithms exhibit relatively stable fluctuation, which is due to the competitive results of increasing number of decomposed subscriptions and decreasing selectivity, as the number of keywords increases. On the other hand, we notice that our index can better utilize the keyword filtering capability than the other competitors when the number of keywords is large (e.g., 6 or 7). Note that we omit the cases when the number of keywords is less than 3, because it is not meaningful to combine less than 3 keywords to generate subscriptions with general boolean expressions. In Fig. 18b, we change the number of subscriptions from 1 to 20 M with the number of keywords being 5, and our AP-Tree structure can scale well with large datasets.

5.2 Experiments for moving subscriptions

In this section, we evaluate the efficiency and effectiveness of AP$^+$-Tree for moving subscriptions.

5.2.1 Experimental setup

For the baseline algorithms, we extend IQ-Tree and RQ-Tree to support moving subscriptions. Specifically, for each location update issued by subscription s, we need to re-evaluate the best cells which can cover s, and re-assign s to these new cells. As to the MsgIndex, we follow the same structure as IQ-Tree and RQ-Tree; that is, for each alive message m, we index m into all the cells which can be penetrated by m. We call these two index variants as IQ$^+$-Tree and RQ$^+$-Tree, respectively. There is no naive way to extend R^t-Tree to support moving subscriptions.

Besides the above two baselines, we also propose two algorithms which index moving subscriptions using grid file, following the common practice in previous work [29,31,43].

- **SS-Index.** SS-Index employs *spatial-prioritized* strategy to build both SubIndex and MsgIndex. For SubIndex, a grid file is first constructed to partition the subscriptions; then in each grid cell, a ranked-key Inverted List is built to further index the subscriptions. MsgIndex follows the similar structure. Note that each message needs to be indexed in all the posting lists which are contained by its keywords.
- **KK-Index.** KK-Index employs *keyword-prioritized* strategy to build both SubIndex and MsgIndex. For SubIndex, a ranked-key Inverted List is built to index subscriptions; then for each posting list, a grid file is built to further partition the subscriptions. MsgIndex follows the similar structure.

We remark that all the four competitors and baselines are implemented using the same incremental evaluation strategy as AP$^+$-Tree, which is illustrated in Algorithm 6. We omit the details due to space limitation.

Datasets We use *TWEETS* and *GN* to verify the performance of our algorithm. The keywords of subscriptions are randomly sampled from *TWEETS* and *GN* following the same strategy in Sect. 5.1.1. In order to generate moving subscriptions, we use the *Network-based Generator of Moving Objects* [4] to generate a set of moving subscriptions. We use the road maps of *Oldenburg* (*ODB* for short, a city in Germany) and *San Joaquin* (*SAN* for short, a county in California) as the input to the generator.[15] The output of the generator is a set of moving points which move on the given road network. We choose these points as the range centers of subscriptions. We set one timestamp as 30 seconds and continuously monitor for 100 timestamps. The parameters are described in Table 3, where the default ones are shown in bold. At each timestamp, there are some subscriptions issuing location update and some new incoming messages. The percentage of moving subscriptions is called *Mobility*. The average velocity of moving subscriptions is set to *medium* by default.[16] The average life cycle of each message is set to 10 timestamps.

Table 3 System parameters for evaluation of moving subscriptions

Parameter	Range
Number of initial subscriptions (M)	0.5, **1**, 1.5, 2.0, 2.5
Number of initial messages (M)	0.5, **1**, 1.5, 2.0, 2.5
Avg. velocity of subscriptions (× *medium*)	0.5, **1**, 2, 3, 4
Subscription mobility (%)	**10**, 20, 40, 60, 80
Arrival rate of new messages in each timestamp (K)	50, **100**, 150, 200, 250
Area of subscription range (%)	0.000001, 0.0001, **0.01**, 1, 10

We use *average Incoming Message Matching* (**IMM** for short) time and *average Moving Subscription Processing* (**MSP** for short) time to evaluate the performance of AP$^+$-Tree. Note that the cost of message insertion is included in **IMM**, unless otherwise specified. The parameters used in the construction of AP$^+$-Tree follow the same setting as Sect. 5.1.1. Besides, we tune the number of grid cells per dimension for both SS-Index and KK-Index, and set it to 40 to balance the performance and memory consumption. We omit the detailed tuning results due to space limitation.

5.2.2 Experimental tuning

Detailed comparison of competitors In this experiment, we compare all the competitive algorithms w.r.t. message insertion time (**MI** for short) and subscriber location update

[15] Thus, we could have four dataset combinations for following experiments: *TWEETS-ODB*, *TWEETS-SAN*, *GN-ODB* and *GN-SAN*.

[16] This is also the default velocity in *Network-based Generator for Moving Objects* [4].

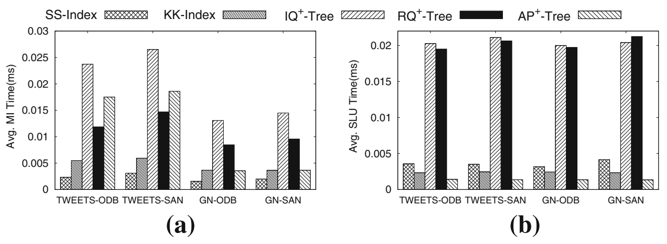


Fig. 19 Detailed comparison between baseline algorithms. **a** Avg. MI time. **b** Avg. SLU time

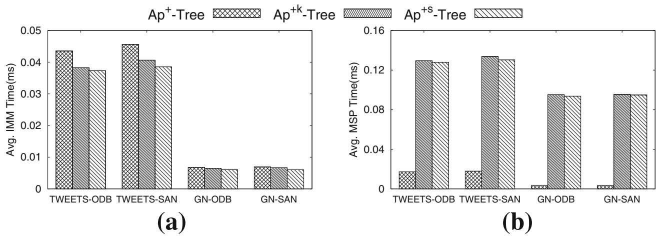


Fig. 20 Comparison of AP^{+}-Tree variants. **a** Avg. IMM time. **b** Avg. MSP time

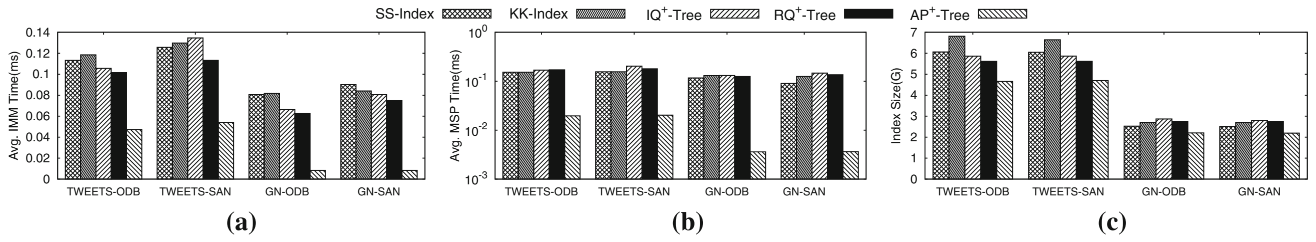


Fig. 21 Evaluation on different datasets. **a** Avg. IMM time. **b** Avg. MSP time. **c** Index size

time (**SLU** for short). It is noticed that IQ^{+}-Tree, RQ^{+}-Tree and AP^{+}-Tree have higher message insertion cost than SS-Index and KK-Index, as shown in Fig. 19a. This is mainly due to the large number of message duplicates in the former three algorithms, thus incurring some additional overhead. However, the message insertion time of our algorithm is still reasonable compared to others. As to the subscriber location update time in Fig. 19b, both IQ^{+}-Tree and RQ^{+}-Tree have undesirable large cost because they have to re-evaluate the optimal cells for each moving subscription. Our index has small update time due to the modified cost model which can penalize s-node with high subscriber location update overhead. Note that in practice, message insertion can be implemented as a by-product of message matching procedure to reduce insertion cost.

Comparison of AP^{+}-Tree variants In this experiment, we verify the effectiveness and efficiency of different AP^{+}-Tree variants. Our variants are designed to separate the SubIndex and MsgIndex; that is, we index subscriptions using AP^{+}-Tree while indexing messages with either *keyword-prioritized* or *spatial-prioritized* indexing structure. We call these two variants as AP^{+k}-Tree and AP^{+s}-Tree, respectively. From Fig. 20a, we notice that AP^{+}-Tree performs a little worse than the other two indexes in terms of incoming message matching, because we build unified index for both subscriptions and messages, thus also considering message insertion time during matching. The benefit of AP^{+}-Tree is shown in Fig. 20b, where AP^{+}-Tree is nearly one order of magnitude faster than the other two indexes w.r.t. moving subscription processing time, thus verifying the effectiveness and efficiency of our proposed unified indexing structure.

5.2.3 Performance evaluation

Evaluation on different datasets We evaluate *average* incoming message matching time, *average* moving subscription processing time and index size against all four datasets in Fig. 21. As shown in both Fig. 21a, b, our algorithm is always the best one in terms of both incoming message matching time and moving subscription processing time. These speedups are gained mainly because of the adaptive organization of moving subscriptions and unified organization of messages. Particularly, for *GN-ODB* and *GN-SAN* datasets, AP^{+}-Tree can achieve up to 30 times improvement for moving subscription processing. Figure 21c indicates that our indexing structure is the most memory efficient, because we organize both subscriptions and messages into a unified structure, thus avoiding the burden to build a separate index for messages.

Effect of mobility In this series of experiments, we evaluate the effect of mobility, i.e., the number of subscriptions moving in each timestamp. Figure 22 reports *total* moving

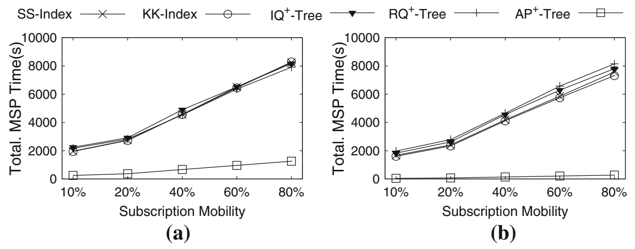


Fig. 22 Effect of mobility. **a** TWEETS-ODB. **b** GN-ODB

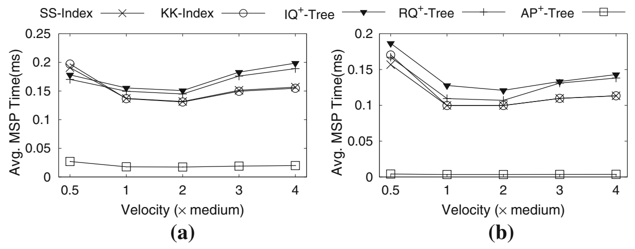


Fig. 23 Effect of velocity. **a** TWEETS-ODB. **b** GN-ODB

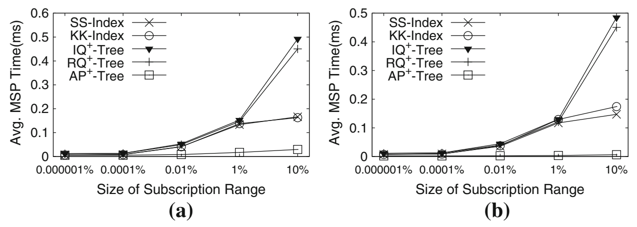


Fig. 24 Effect of subscription range size. **a** TWEETS-ODB. **b** GN-ODB

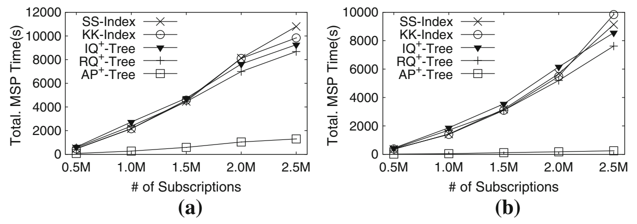


Fig. 25 Effect of number of subscriptions. **a** TWEETS-ODB. **b** GN-ODB

subscription processing time as we increase mobility from 10 to 80 %. We do not report incoming message matching time as it is not sensitive to the change of mobility. All the results demonstrate that, as mobility increases, our index can always beat the baselines with a large margin. This is mainly due to the benefit of our modified cost model, which can be adaptive to the movement of subscriptions.

Effect of moving velocity We evaluate the influence of velocity w.r.t. the moving subscription processing time in this experiment. Figure 23 shows the results against *TWEETS-ODB* and *GN-ODB* datasets, where we increase the speed from 0.5 to 4 times of the default *medium* velocity. We observe that subscriptions are more *selective* and can match more messages when velocity is slow, leading to large processing time at initial. However, as velocity increases, the *selectivity* decreases due to the change of underlying spatial distribution, resulting in the decrease in processing time. Finally, when we increase velocity further, the effectiveness of incremental evaluation technique degenerates, thus leading to the increase in processing time. In all the cases, AP$^+$-Tree can always achieve best performance.

Effect of subscription range size In this set of experiments, we evaluate the performance of our techniques when changing the subscription range size from 0.000001 to 10 % of the data space against *TWEETS-ODB* and *GN-ODB* datasets in Fig. 24. As shown in the figure, the moving subscription processing time increases for all algorithms when increasing subscription range, because more messages will be covered by large range. However, the processing time of our algorithm is much smaller than competitors and increases much slower as well.

Effect of number of subscriptions In this series of experiments, we evaluate the scalability of the proposed techniques in Fig. 25. We increase the number of initial subscriptions and messages from 0.5 to 2.5 M. The message arrival rate also increases accordingly. Figure 25a, b show moving subscription processing time. All the results indicate our algorithm can scale well with increasing dataset size, thus making it practical to implement our algorithm into a large-scale real system.

Comparison with Elaps In this set of experiments, we compare our methods with Elaps, which is recently proposed in [20] on *TWEETS-ODB* dataset. For fair comparison, we follow the similar setting in [20]. The initial number of subscriptions and messages are set to 10 and 1 M, respectively. The velocity, mobility and arrival rate of new messages are set to 1 × *medium*, 100 % and 10 K, respectively. The source code of Elaps is kindly provided by authors in [20]. Please note that in this set of experiments, the spatial regions of subscriptions are represented by circles to be consistent with Elaps. For completeness, we also propose a new index, called AP$^+$-Tree-Safe, which integrates into AP$^+$-Tree a light-weighted version of the safe region technique proposed in Elaps.[17] We compare the total processing time and communication cost of the above three algorithms. In Fig. 26, we vary the number of subscriptions from 10 to 1 M. It is observed that AP$^+$-Tree is orders of magnitude faster than Elaps w.r.t. total processing time (Fig. 26a), while Elaps wins w.r.t. communication cost (Fig. 26b). It is also notice-

[17] The light-weighted version regards the distance from the subscription to its nearest matching message minus the search radius of the subscription as the safe region radius. The impact region is derived by extending the safe region with the search radius of subscription.

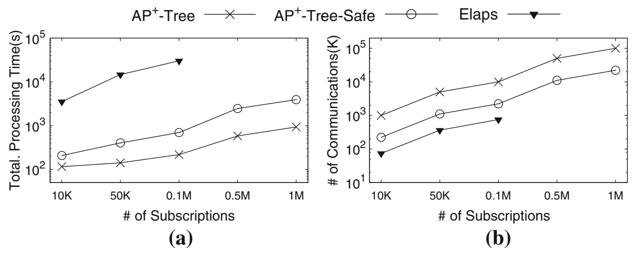


Fig. 26 Effect of number of subscriptions. **a** TWEETS-ODB. **b** TWEETS-ODB

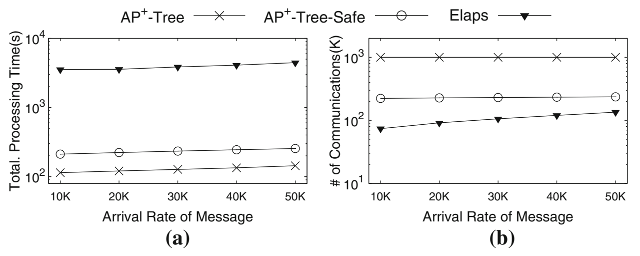


Fig. 27 Effect of message arrival rate. **a** TWEETS-ODB. **b** TWEETS-ODB

able that when the number of subscriptions is larger than 0.1 M, Elaps returns *run out of memory error*, while our index can scale to 1 M subscriptions easily. On the other hand, our new index AP^+-Tree-Safe can achieve a good trade-off between computation cost and communication cost, which is desirable for real-life applications. In Fig. 27, we vary the arrival rate of messages from 10 to 50 K, and the similar trends can be observed.

6 Conclusion

The phenomenon of streaming spatial-textual data raises interesting challenges for indexing continuous spatial-keyword subscriptions. In this paper, we propose a novel adaptive spatial-textual partition indexing structure, namely AP-Tree, to efficiently organize a massive number of spatial-keyword subscriptions such that each incoming message from spatial-textual data stream can be rapidly delivered to relevant subscriptions. Unlike the previous spatial-textual indexes which prefer either textual feature or spatial feature, AP-Tree can be constructed in an adaptive way by carefully choosing keyword or spatial partitions guided by a cost model. Furthermore, we extend our indexing structure to support moving spatial-keyword subscriptions, following the same framework as AP-Tree while with a modified cost model and a unified organization for both subscriptions and messages. Extensive experiments demonstrate that our techniques for both stationary and moving subscriptions achieve a high-throughput performance over spatial-textual stream.

Acknowledgments Ying Zhang is supported by ARC DE140100679 and DP130103245. Wenjie Zhang is supported by ARC DP150103071 and DP150102728. Xuemin Lin is supported by NSFC61232006, ARC DP150102728, DP140103578 and DP120104168. Wei Wang is supported by ARC DP130103401 and DP130103405.

References

1. Aguilera, M.K., Strom, R.E., Sturman, D.C., Astley, M., Chandra, T.D.: Matching events in a content-based subscription system. In: PODC, pp. 53–61 (1999)
2. Bao, J., Mokbel, M.F., Chow, C.Y.: Geofeed: A location aware news feed system. In: ICDE, pp. 54–65 (2012)
3. Bentley, J.L.: Solutions to Klees Rectangle Problems. Technical report, Carnegie-Mellon University, Pittsburgh, PA (1977)
4. Brinkhoff, T.: A framework for generating network-based moving objects. GeoInformatica **6**(2), 153–180 (2002)
5. Cheema, M.A., Brankovic, L., Lin, X., Zhang, W., Wang, W.: Multi-guarded safe zone: an effective technique to monitor moving circular range queries. In: ICDE (2010)
6. Chen, L., Cong, G., Cao, X.: An efficient query indexing mechanism for filtering geo-textual data. In: SIGMOD, pp. 749–760 (2013)
7. Chen, L., Cong, G., Cao, X., Tan, K.: Temporal spatial-keyword top-k publish/subscribe. In: ICDE, pp. 255–266 (2015)
8. Chen, L., Cong, G., Jensen, C.S., Wu, D.: Spatial keyword query processing: an experimental evaluation. In: PVLDB, pp. 217–228. VLDB Endowment (2013)
9. Chen, X., Chen, Y., Rao, F.: An efficient spatial publish/subscribe system for intelligent location-based services. In: DEBS (2003)
10. Christoforaki, M., He, J., Dimopoulos, C., Markowetz, A., Suel, T.: Text vs. space: efficient geo-search query processing. In: CIKM (2011)
11. Cong, G., Jensen, C.S., Wu, D.: Efficient retrieval of the top-k most relevant spatial web objects. PVLDB **2**(1), 337–348 (2009)
12. De Berg, M., Van Kreveld, M., Overmars, M., Schwarzkopf, O.C.: Computational Geometry. Springer, Berlin (2000)
13. De Felipe, I., Hristidis, V., Rishe, N.: Keyword search on spatial databases. In: ICDE, pp. 656–665 (2008)
14. Du, X., Li, Y., Huang, Q., Shu, L.: Search continuous spatial keyword range queries over moving objects in road networks. J. Comput. Inf. Syst. **11**(2), 759–767 (2015)
15. Enderton, H., Enderton, H.B.: A mathematical Introduction to Logic. Academic Press, Waltham (2001)
16. Fabret, F., Jacobsen, H.A., Llirbat, F., Pereira, J., Ross, K.A., Shasha, D.: Filtering algorithms and implementation for very fast publish/subscribe. In: SIGMOD, pp. 115–126 (2001)
17. Fabret, F., Jacobsen, H.A., Llirbat, F., Pereira, J., Ross, K.A., Shasha, D.: Filtering algorithms and implementation for very fast publish/subscribe systems. In: ACM SIGMOD Record, vol. 30, pp. 115–126. ACM (2001)
18. Grigni, M., Manne, F.: On the complexity of the generalized block distribution. In: Parallel Algorithms for Irregularly Structured Problems, pp. 319–326 (1996)
19. Guo, L., Shao, J., Aung, H.H., Tan, K.: Efficient continuous top-k spatial keyword queries on road networks. GeoInformatica **19**(1), 29–60 (2015)
20. Guo, L., Zhang, D., Li, G., Tan, K., Bao, Z.: Location-aware pub/sub system: when continuous moving queries meet dynamic event streams. In: SIGMOD, pp. 843–857 (2015)
21. Helmer, S., Moerkotte, G.: A performance study of four index structures for set-valued attributes of low cardinality. VLDB J. **12**, 244–261 (2003)
22. Hmedeh, Z., Kourdounakis, H., Christophides, V., Du Mouza, C., Scholl, M., Travers, N.: Subscription indexes for web syndication systems. In: EDBT, pp. 312–323 (2012)

23. Hu, H., Liu, Y., Li, G., Feng, J., Tan, K.: A location-aware publish/subscribe framework for parameterized spatio-textual subscriptions. In: ICDE, pp. 711–722 (2015)
24. Huang, W., Li, G., Tan, K.L., Feng, J.: Efficient safe-region construction for moving top-k spatial keyword queries. In: CIKM (2012)
25. Konig, A., Church, K., Markov, M.: A data structure for sponsored search. In: ICDE, pp. 90–101 (2009)
26. Kullback, S.: Information Theory and Statistics. Courier Dover Publications, Mineola (1997)
27. Li, C., Gu, Y., Qi, J., Yu, G., Zhang, R., Yi, W.: Processing moving k NN queries using influential neighbor sets. Proc. VLDB **8**(2), 113–124 (2014)
28. Li, G., Wang, Y., Wang, T., Feng, J.: Location-aware publish/subscribe. In: SIGKDD, pp. 802–810 (2013)
29. Mokbel, M.F., Xiong, X., Aref, W.G.: Sina: Scalable incremental processing of continuous queries in spatio-temporal databases. In: SIGMOD, pp. 623–634 (2004)
30. Mouratidis, K., Pang, H.: Efficient evaluation of continuous text search queries. TKDE **23**(10), 1469–1482 (2011)
31. Mouratidis, K., Papadias, D., Hadjieleftheriou, M.: Conceptual partitioning: an efficient method for continuous nearest neighbor monitoring. In: SIGMOD (2005)
32. Nutanong, S., Zhang, R., Tanin, E., Kulik, L.: The v*-diagram: a query-dependent approach to moving KNN queries. Proc. VLDB **1**(1), 1095–1106 (2008)
33. Park, M.H., Hong, J.H., Cho, S.B.: Location-based recommendation system using bayesian users preference model in mobile devices. In: Ubiquitous Intelligence and Computing, pp. 1130–1139. Springer (2007)
34. Rocha-Junior, J.B., Gkorgkas, O., Jonassen, S., Nørvåg, K.: Efficient processing of top-k spatial keyword queries. In: SSTD (2011)
35. Sadoghi, M., Jacobsen, H.A.: Be-tree: an index structure to efficiently match boolean expressions over high-dimensional discrete space. In: SIGMOD (2011)
36. Samet, H.: The quadtree and related hierarchical data structures. ACM Comput. Surv. **16**(2), 187–260 (1984)
37. Shraer, A., Gurevich, M., Fontoura, M., Josifovski, V.: Top-k publish-subscribe for social annotation of news. Proc. VLDB **6**, 385–396 (2013)
38. Swami, A.N.: Optimization of large join queries: combining heuristic and combinatorial techniques. In: SIGMOD, pp. 367–376 (1989)
39. Terrovitis, M., Bouros, P., Vassiliadis, P., Sellis, T.K., Mamoulis, N.: Efficient answering of set containment queries for skewed item distributions. In: EDBT, pp. 225–236 (2011)
40. Wang, X., Zhang, Y., Zhang, W., Lin, X., Wang, W.: Ap-tree: Efficiently support continuous spatial-keyword queries over stream. In: ICDE (2015)
41. Whang, S.E., Garcia-Molina, H., Brower, C., Shanmugasundaram, J., Vassilvitskii, S., Vee, E., Yerneni, R.: Indexing boolean expressions. Proc. VLDB **2**(1), 37–48 (2009)
42. Wu, D., Yiu, M.L., Jensen, C.S., Cong, G.: Efficient continuously moving top-k spatial keyword query processing. In: ICDE (2011)
43. Xiong, X., Mokbel, M.F., Aref, W.G.: Sea-cnn: Scalable processing of continuous k-nearest neighbor queries in spatio-temporal databases. In: ICDE, pp. 643–654 (2005)
44. Xu, W., Chow, C.Y., Yiu, M.L., Li, Q., Poon, C.K.: Mobifeed: a location-aware news feed framework for moving users. GeoInformatica **19**, 1–37 (2014)
45. Yan, T.W., García-Molina, H.: Index structures for selective dissemination of information under the boolean model. TODS (1994)
46. Yan, T.W., Garcia-Molina, H.: Duplicate removal in information system dissemination. In: PVLDB (1995)
47. Zhang, C., Zhang, Y., Zhang, W., Lin, X.: Inverted linear quadtree: efficient top k spatial keyword search. In: ICDE, pp. 901–912. IEEE (2013)
48. Zhang, D., Chan, C.Y., Tan, K.L.: An efficient publish/subscribe index for e-commerce databases. Proc. VLDB **7**(8), 613–624 (2014)
49. Zhang, J., Zhu, M., Papadias, D., Tao, Y., Lee, D.L.: Location-based spatial queries. In: SIGMOD, pp. 443–454 (2003)
50. Zhou, Y., Xie, X., Wang, C., Gong, Y., Ma, W.Y.: Hybrid index structures for location-based web search. In: CIKM, pp. 155–162 (2005)