Indexing Evolving Events from Tweet Streams

Hongyun Cai, Zi Huang, Divesh Srivastava, and Qing Zhang

Abstract—Tweet streams provide a variety of real-life and real-time information on social events that dynamically change over time. Although social event detection has been actively studied, how to efficiently monitor evolving events from continuous tweet streams remains open and challenging. One common approach for event detection from text streams is to use single-pass incremental clustering. However, this approach does not track the evolution of events, nor does it address the issue of efficient monitoring in the presence of a large number of events. In this paper, we capture the dynamics of events using four event operations (create, absorb, split, and merge), which can be effectively used to monitor evolving events. Moreover, we propose a novel event indexing structure, called Multi-layer Inverted List (MIL), to manage dynamic event databases for the acceleration of large-scale event search and update. We thoroughly study the problem of nearest neighbour search using MIL based on upper bound pruning, along with incremental index maintenance. Extensive experiments have been conducted on a large-scale real-life tweet dataset. The results demonstrate the promising performance of our event indexing and monitoring methods on both efficiency and effectiveness.

Index Terms—Event indexing, multi-layer inverted list, event evolution

1 Introduction

As one of the most popular online social networking services, Twitter has been witnessing a burst of growth in the numbers of both users and posts since it was created in March 2006. The quickly-updated tweets cover a wide variety of events that happen around the world everyday. These events reveal valuable information on breaking news, hot discussions, public opinions, and so on. Moreover, these events typically evolve over time. Event evolution exhibits event changes across successive time stamps. For example, public opinions about a US presidential election may change over time. Its evolution can unfold a history of the event and characterize the event with changing opinions over time, providing an opportunity for timely responses at different stages. Thus how to efficiently and effectively monitor event evolution from tweet streams is of great importance.

The characteristics of tweets pose two main challenges to event evolution monitoring (EEM). First, the textual content of a tweet is short and noisy, which may affect the quality of event tracking. Second, tweets keep arriving in a streaming fashion. According to the online statistics, there are around 58 million tweets posted on Twitter per day on average. The event monitoring algorithms for such kind of dynamic social data have to be scalable and incremental without a priori knowledge. Existing event detection methods, such as the single-pass incremental clustering (SPIC) algorithm [1], [2], [3], can hardly be applied to event evolution monitoring due to the following two limitations. First, once a cluster is created, it can only be updated by inserting new tweets which is too restrictive to capture the evolution of events, which may require merging and splitting of existing clusters. Second, it lacks indexing support. It needs to compare the newly arriving tweet with all existing events exhaustively. Without an index, the performance will deteriorate when the data size and the number of events grow quickly. However, indexing structures which are designed for long text documents such as Web pages cannot efficiently process short and rapidly arriving tweets. Most of the existing studies on event evolution monitoring focus on tracking the details of one event (e.g., [1], [4]). The peaks of the tweet activities are highlighted by analysing the tweets collection belonging to a specific event, but the relationships among multiple events and the changes are missed in their work. To monitor evolutions among all relevant events, a subgraph-by-subgraph incremental tracking framework is proposed in [5]. Six primitive cluster evolution operations are defined to capture the dynamic changes of events efficiently. However, they only check the evolution of events when the time window moves and thus fail to monitor event evolutions in real time. Moreover, the setting of the time window length could be problematic. More details are discussed in Section 2.1.

In view of the lack of effective methods for monitoring evolving events from tweet streams, we use four event operations to capture dynamic event evolution patterns, including creation, absorption, split and merge. These four operations are able to track event evolution over time. Further, split and merge operations can also record event relationships in the evolution process. Although similar operations have been mentioned in previous work [6], our work focuses on how to support these operations efficiently by our proposed indexing structure (MIL). To implement fast event search upon the arrival of new tweets, we propose a Multi-layer Inverted List (MIL) as an event indexing structure that can support both efficient event search (for the four event operations) and real-time event update. More specifically, a multi-layer structure is constructed based on

- *H. Cai is with the School of ITEE, The University of Queensland, Australia, and CSIRO ICT Centre, Australia. E-mail: h.cai2@uq.edu.au.*
- *Z. Huang is with the School of ITEE, The University of Queensland, QLD 4072, Australia. E-mail: huang@itee.uq.edu.au.*
- *D. Srivastava is with the AT&T Labs-Research, Bedminster, NJ 07921, USA. E-mail: divesh@research.att.com.*
- *Q. Zhang is with the CSIRO ICT Centre, Herston, QLD 4029, Australia. E-mail: qing.zhang@csiro.au.*

Manuscript received 9 Jan. 2015; revised 5 May 2015; accepted 28 May 2015. Date of publication 15 June 2015; date of current version 2 Oct. 2015. Recommended for acceptance by C. Chan. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TKDE.2015.2445773

the traditional inverted list indexing to support fast search and update for large-scale event databases. By exploiting the strong correlations among words in short tweets, the mth layer in the structure contains the inverted lists for m-terms (i.e., a sequence of m words sorted in alphabetical order that frequently co-occur in tweets). It is designed in a way that more relevant yet shorter event lists are quickly found at the lowest layer, hence searching longer event lists at the upper layer can be largely avoided by our proposed pruning strategy. Note that MIL is an indexing structure with root (the first layer) at the top, hence upper layer is closer to the root of MIL while lower layer is closer to the leaf. In other words, equipped with the MIL indexing structure, nearest neighbour search is implemented with effective upper bound pruning. A novel upper bound on event similarity is designed to significantly reduce the computational cost on the long event lists.

The contributions of this paper are summarized as follows:

- We use four event operations along with the stream clustering algorithm to effectively capture event evolution patterns over time.
- We propose a novel event indexing structure, referred to as Multi-layer Inverted List, to facilitate event search and event update for large-scale, dynamic event databases.
- We present efficient algorithms for nearest neighbour search with upper bound pruning to avoid a large proportion of expensive event similarity computations.
- We conduct an extensive performance study on dynamic event databases generated from over 10 million tweets and the results demonstrate the superiority of our methods over existing methods.

The remainder of this paper is organized as follows. Section 2 reviews related work. Section 3 formally defines the problem. Section 4 details event evolution monitoring strategies. MIL and its search algorithms are discussed in Section 5. Experimental results are shown in Section 6 and we conclude in Section 7.

2 Related Work

In this section, we review related work on two topics: event detection/tracking in Twitter and text indexing, and then present the differences between existing work and ours at the end.

2.1 Event Detection and Tracking in Twitter

Event detection. Recently, considerable efforts have been devoted to summarizing online textual streams such as tweets in the form of events [7], [8], [9], [10]. There are various ways to give a taxonomy of all these related works. As shown in [11], according to the *detection task*, the techniques can be classified into retrospective event detection (RED) [12] and new event detection (NED) [13]. Based on the *event type*, they are categorized into specified [14] and unspecified [15] event detection. The former relies on the prior available information on the event of interest, while the latter detects events from the bursts or trends in Twitter streams. Depending on the *detection method*, there are supervised [16] and unsupervised [17] algorithms. In this paper, we focus on NED, unspecified and unsupervised event detection. We undertake a brief review of several representative studies in the same category. To capture emerging events, [18] identifies the locally dense sub-graphs from a graph under a streaming model. In [17], daily signals for each word are constructed by applying wavelet analysis. The words are clustered into events using a modularity-based graph partitioning algorithm. This kind of daily detected events lose track of their developments over multiple days. In [19], the single-pass incremental clustering algorithm is adopted for event detection and the threshold is dynamically generated by the statistics of existing clusters. However, the influence of cluster center shift is ignored, thus it fails to capture the dynamics of events.

Event evolution. Most of the work on event evolution tracks the details of one specified event in real time. For example, in [1], the authors track the representative tweets of an event and mine geographical diffusion trajectory on the map. Instead of tracking one individual event, we focus on event evolution, aiming to monitor the event evolution among relevant events. To the best of our knowledge, [5] is the only effort to monitor event evolution. In their work, a subgraph-by-subgraph incremental tracking framework is proposed for event monitoring. A skeletal graph is designed to summarize the information within a fading time window in a dynamic network. However, the evolution graph is constructed by treating each event snapshot as a node and the trajectory between snapshots as paths. So the evolution of an event is only checked when the time window moves. This time window strategy faces the problem that long time window length may lead to losing track of highly dynamic events' evolutions and short time window length will lead to storing redundant snapshots for steady events.

2.2 Text Indexing

Traditional text indexing. Text indexing is an important area and has been widely studied to facilitate textual information retrieval in various information systems. Among all the indexing structures, the inverted file has proven to be the most effective structure for text similarity search [20]. The relevant research has been explored in many directions, including index construction, index maintenance, index storage, index compression, etc. We are particularly interested in index construction and maintenance. The traditional inverted file maintains a list for each word in the vocabulary, where each list contains the identifiers of the documents that contain the word and some other advanced information such as the in-document term frequencies. Different variants have been proposed to improve the performance. Phrase index (e.g., [21]) stores word sequences rather than individual words as the index terms to support fast phrase queries. But the word sequence in phrase index is a phrase which is an ordered list of words in natural language while our index term is a set of frequently co-occurring words sorted in alphabetical order. Pitts et al. [22] proposed a two-layer indexing structure over the partitions of author names and on the clusters inside each partition to efficiently retrieve an author query to its closest cluster for

duplicate detection. Their indexing structure is constructed based on the offline cluster results while ours is used to accelerate the processing of online clustering. The most relevant work to ours is the integration of the trie-tree structure and inverted file to index set-valued attributes for superset, subset and equality queries [23]. The key idea is to store the frequent items as a trie-tree structure in main memory, along with the corresponding inverted sublists. Hence the IO cost of transferring the disk pages with the inverted lists to main memory is reduced. Our work is complementary to theirs, since their index structure is designed to reduce the IO cost, while our in-memory index structure stores the upper bound information and provides tighter upper bounds in lower layers to prune the long retrieved lists in upper layers so as to reduce the similarity computation cost.

Indexing for Twitter. Unlike traditional text indexing, the indexing structure for Twitter should be capable of ingesting the data rapidly and support efficient search and quick update on large-scale data. In [24], a retrieval engine which supports Twitter's real-time search is introduced. The authors focus on how to organize the inverted list indexing to support low-latency, high-throughput retrieval and how to implement concurrency management. In [25], a query-based classification approach is designed to distinguish between tweets that may appear as a search result with high probability and the noisy tweets. The former will be indexed by the inverted list in real-time, and the latter will be indexed in a background batch scheme. As we can see, all these indexing structures are constructed to index tweets and the search algorithms are designed to support fast tweet search on tweets data not event search on the detected events (clusters of tweets). The difference is that only a limited number of words exist in one tweet (even with weights) and the tweets search uses a Boolean query language, i.e., search tweets containing (or not containing) specified search term(s); while for events, there might be hundreds of words in one event and only a few of them are dominant words (i.e., words with much higher weight than others). Moreover, event search is nearest neighbour search which is more complex than Boolean queries. To the best of our knowledge, only one event indexing structure named variable dimensional extendible hash (VDEH) [26] has been proposed before. Given a query tweet, they use the fraction of matched tweets in an event to calculate the similarity between a query tweet and an event. VDEH stores highly similar tweets together in one bucket of the hash to accelerate the comparison between the query tweet and each event. Besides indexing, another approach for improving efficiency is the distributed processing topology [27]. However, for the highly dynamic Twitter data stream, detecting events immediately after the tweets are posted is of great importance. Unlike the distributed system, our proposed in-memory indexing structure can process data in real time by avoiding disk IO.

Our work in this paper can be distinguished from previous work in several ways. First, in addition to event detection, we aim to track evolving events over time. Unlike previous work which checks the evolution patterns only when the time window moves, we record the evolution patterns and identify event relationships whenever events evolve. Second, we design an indexing structure for event databases to support event search and update. Although VDEH has been proposed to index events, the actual data indexed by VDEH contains all the tweets in each event, which leads to large storage cost. In contrast, ours is an in-memory indexing structure which stores only the summary representation of the event. By avoiding the IO cost, we are able to process the highly dynamic tweets data in real time. Third, we design new and tight upper bounds on the Cosine similarity between tweets and events to quickly reduce the search space for nearest neighbour search. Fourth, existing work on indexing tweets mostly focuses on organizing the quickly incoming tweets while our work aims to improve the performance of searching the detected events given tweets.

3 Problem Formalization

In this section, we will introduce some basic concepts used in our paper and the research problem we aim to address.

Tweet. A tweet is a small piece of text, which is usually represented by the vector space model [3]. The augmented term frequency [28] is applied to our representation because it has been proven to show the best performance in representing dynamic, short and noisy tweets [29]. Assuming the vocabulary used in our vector space model is a bag of words $\{w_1, w_2, \ldots, w_n\}$, we represent a tweet as below.

Definition 1. *Given a tweet e, it is represented by a vector $< tf(w_1), tf(w_2), \ldots, tf(w_n) >$, where $tf(w_i)$, the value of the ith dimension of e, indicates the augmented term frequency of the corresponding word w_i in e, which is calculated as*

$$tf(w_i) = \begin{cases} 0.5 + \frac{0.5 \times f(w_i,e)}{\max\{f(w,e):w \in e\}}, & f(w_i,e) > 0 \\ 0, & f(w_i,e) = 0 \end{cases} \quad (1)$$

where $f(w_i, e)$ is the term frequency of w_i in e, $i = 1..n$, and n is the vocabulary size. $tf(w_i) = 0$ if w_i does not appear in e.

Event. An event is defined as a cluster of tweets sharing similar textual information.

Definition 2. *Given a similarity metric φ[1] and a similarity constraint θ, an event E consisting of a set of tweets $\{e1, e2, \ldots\}$ needs to satisfy the following condition that*

$$\forall e \in E \rightarrow \varphi(e, E.center) \geq \theta,$$

where $E.center$ is defined as the mean vector of all tweet vectors belonging to E, in which each value is the mean of the augmented term frequencies of the corresponding word.

We simply represent an event E by its center vector $E.center$, which also applies the vector space model. Thus, the constraint θ can be considered as the smallest acceptable similarity between the event and its member tweets. A number of important event properties are also recorded, which are described in Table 1. We assume that there is no overlap among events. In other words, any single tweet only belongs to one event. This is reasonable because of the small lengths of tweets. More notations related to E used in the following sections are listed in Table 2.

The problem. The problem we aim to address in this paper is to effectively and efficiently detect emerging events,

1. *Cosine* similarity is used in our work.

TABLE 1
Event Properties

Property	Notation	Description
Size	$\|E\|$	the number of tweets in E
Radius	r_E	the smallest similarity from the tweets in E to the cluster center
Starting time	ts_E	the initial creation time of E
Evolving period	te_E	the time duration of E from its creation to the latest update
Previous events	$prev_E$	the events where E is derived from
Next events	$next_E$	the events derived from E

capture the changes of existing events and eventually reveal events' evolution paths over continuous tweet streams. Given a tweet stream, we propose to incrementally generate a set of events $S = \{E1, E2, \ldots\}$ along the timeline and analyse the evolving relationships among them, e.g., which event is the ancestor of another one. Four operations are defined in our work to describe the evolving patterns: creation, absorption, split and merge. The evolving paths are recorded by the event properties $prev_E$ and $next_E$ (see Table 1). For example, if $E1$ splits into $E2$ and $E3$, we will set $next_{E1} = \{E2, E3\}$, $prev_{E2} = E1$ and $prev_{E3} = E1$.

4 Monitoring Evolving Events

Social events are highly dynamic and evolving quickly. It is important to capture their dynamics and record their evolutions, which help to make sense of the influence and the trend of social events. In this section, we will define four event evolution operations, based on which evolving events are captured incrementally.

4.1 Event Evolution Operations

We define four event evolution operations including creation, absorption, split and merge, to reflect event changes upon the arrival of new tweets. Given a set of N existing events $\{Ej|j = 1..N\}$ and a new arriving tweet e, we first find the most similar event to e according to the predefined similarity metric φ and denote it as E_{NN}. Based on the similarity constraint θ on the events, the following operations are triggered by the arrival of e (similar to the operations in [6]).

Create: If $\varphi(e, E_{NN}) < \theta$, the similarity constraint on events is not satisfied. Thus, a new event is created, which consists of the single e.

Absorb: If $\varphi(e, E_{NN}) \geq \theta$, the new tweet e is supposed to be absorbed by E_{NN}. However, when e joins E_{NN}, the event center may shift, resulting in the change of the smallest similarity between the member tweets and the event. If the updated smallest similarity value still satisfies the constraint θ, i.e., not smaller than θ, E_{NN} will be updated by absorbing e. Otherwise, E_{NN} will no longer be a valid event and the operation *Split* will be triggered.

Split: If the updated smallest similarity value between the member tweet and E_{NN} is smaller than θ by absorbing e, the bisecting k-Means clustering [30] will be performed on the tweets contained by E_{NN} to generate two new events.

TABLE 2
Notations

Notation	Description
Ej	the jth event
E_i	the ith dimension in the vector representation of E
$\overline{E_i}$	the ith largest value among all dimensions of E
$\overline{Ej_i}$	the ith largest value among all dimensions of Ej

Merge: When the operation split is conducted, the newly split events may be merged with other existing events if the merged events satisfy the similarity constraint θ. In this case, the split events need to check whether any existing events can be merged, and the merged events are regarded as new events.

The time complexity to adjust the event center when absorbing a new tweet is linear to the number of words in the updated event. Obviously, split and merge operations indicate the evolving relationships among events. Whenever a split or merge operation happens, event properties $prev_E$ and $next_E$ of new events are updated accordingly, which record the event evolution path and reveal how events are evolving over time.

To give a better explanation of the above four operations for monitoring evolving events, we will use a real-life example, i.e., Edward Snowden's leakage of NSA documents. When Snowden just left Hawaii for Hong Kong, a lot of tweets were posted to discuss him. These tweets form the event "$E1$ (Snowden, NSA, Hong Kong, USA)".[2] After Snowden flew to Russia, the focus of new tweets changed as well. At the beginning, these tweets were absorbed by $E1$ because they were close to $E1$ by sharing a lot of common words like "Snowden", "Edward", and "USA". However, when more and more new tweets were involved, the smallest similarity between tweets and the event center of $E1$ was getting smaller and smaller due to the shifting focus of the new arriving tweets, which made the content of the event $E1$ more and more diverse. Eventually, the focus of $E1$ changed from "Hong Kong" to "Hong Kong, Russia". At one moment, $E1$ did not satisfy the similarity constraint θ anymore after receiving the new tweets. Hence $E1$ was split into "$E2$ (Snowden, Russia, USA, Putin)" and "$E3$ (Snowden, Hong Kong, NSA, leak)". Once the split operation was triggered, the newly split events (i.e., $E2$ and $E3$) could be merged with other nearby events. For example, $E2$ was merged with "$E4$ (Russia, USA, Obama, Putin)" to generate the new event $E5$ which satisfied the threshold requirement.

The only parameter in defining these four event evolution operations is the similarity constraint θ. Instead of predefining θ, we aim to automatically adjust it according to the statistics of previous query tweets' nearest neighbours' similarity records. The idea is to maintain the mean μ and standard deviation σ of all the previous query tweets' largest similarity values to the database events. We follow the settings in [19], and the threshold is dynamically calculated as $\mu - 3\sigma$. Statistically, the 3σ range to the mean covers more than 99.7 percent tweets in the event for normally

2. Note that, rather than using formal vector space model, in this example, we only use four most frequent words of an event as its representatives for illustration purpose.

distributed tweets. That is to say, when the similarity between a new tweet and its nearest neighbour event is smaller than θ, it is highly unlikely for the tweet to be relevant to its most similar event and a new event should be created. The mean and standard deviation of the largest similarity values of previous query tweets can be maintained dynamically by their zero'th, first and second order moments continuously, which corresponds to the number of tweets, the sum of their largest similarity values, and the sum of squared largest similarity values respectively. They can be additively calculated over the tweet stream, thus are easily maintained in the streaming scenario. Denoting the three moments as M_0, M_1, and M_2 respectively, the mean μ and standard deviation σ can be obtained by these moments as $\mu = M_1/\ M_0$ and $\sigma = \sqrt{M_2/M_0 - (M_1/M_0)^2}$.

Algorithm 1. Process a New Tweet

Input: Events set S, New tweet e, Zero'th, first and second order moments M_0, M_1, M_2, Threshold upper bound θ_U
Output: S, M_0, M_1, M_2

1 **if** $S = \emptyset$ **then**
2 S.add(CreateEvent(e));
3 $M_0 = M_1 = M_2 = 0$;
4 **else**
5 $E_{NN}, \varphi_{max} \leftarrow$ NearestNeighbourSearch(S, e) ;
6 $\theta \leftarrow \frac{M_1}{M_0} - 3 \times \sqrt{\frac{M_2}{M_0} - \left(\frac{M_1}{M_0}\right)^2}$;
7 **if** $\theta > \theta_U$ **then** $\theta \leftarrow \theta_U$;
8 **if** $\varphi_{max} < \theta$ **then**
9 S.add(CreateEvent(e));
10 **else**
11 E_{NN}.AbsorbEvent(e);
12 **if** $r_{E_{NN}} < \theta$ **then**
13 $E1, E2 \leftarrow E_{NN}$.SplitEvent();
14 S.remove(E_{NN});
15 S.MergeEvent(θ, $E1$, $E2$);
16 **end**
17 **end**
18 **end**
19 $M_0 \leftarrow M_0 + 1$;
20 $M_1 \leftarrow M_1 + \varphi_{max}$;
21 $M_2 \leftarrow M_2 + \varphi_{max} \times \varphi_{max}$;
22 **return** S, M_0, M_1, M_2;

Besides the automatically generated threshold θ, we also define θ_U, an upper bound of θ to prevent an unnecessarily high θ value. For example, the burst of a hot event will lead to a large number of high similarity values to the event. This will make the value of θ keep increasing, and other less intensive events will fail to absorb relevant tweets due to the extremely high value of the threshold setting. θ_U can help keep θ values in a reasonable range by filtering the outliers. The process of tuning θ_U is detailed in Section 6.2. We do not worry about the low θ value because θ generally goes up as new tweets are inserted.

4.2 Incremental Event Evolution

Based on the above four event evolution operations, the events will be updated with the streaming arrivals of tweets. Two algorithms are designed in our work to implement the four operations and update the events dynamically. The whole process is described in Algorithm 1.

Algorithm 1 starts with an empty event set S. For the very first tweet, it is taken as an event to be added into the event set (or database), followed by initializing order moments (lines 1-3). For any of the following tweets, its nearest neighbour event and the corresponding similarity φ_{max} is first found by conducting the nearest neighbour search on the events set (line 5). The threshold θ is then dynamically computed based on M_0, M_1 and M_2 (line 6). If θ is greater than the upper bound θ_U, θ is updated to be θ_U (line 7). If the nearest neighbour event E_{NN} has smaller similarity to e than θ, a new event is created and added into the event set (lines 9). Otherwise, e is absorbed by E_{NN} (line 11). If the updated radius of E_{NN} is smaller than θ, E_{NN} is split into two new and smaller events by adopting the bisecting k-Means clustering [30], followed by merging events (lines 13-15). Finally, three moments are updated incrementally (lines 19-21).

Algorithm 2 presents the steps to check whether the split events $E1$ and $E2$ should be merged with similar events. Given a split event E, the events with the Cosine similarity to it higher than θ are retrieved and added into a candidates set which are sorted by their similarities with E for merge operation (line 2). Then these candidate events are merged with E one by one (line 3-16) and the smallest similarity between the member tweets and the new merged event is checked every time (line 5). The merge procedure for E will stop when the new merged event does not satisfy the similarity constraint θ of forming an event (lines 5-7) or when all events in the candidate set are merged (lines 13-15). When the merge procedure finishes, event properties are updated based on the merged event.

Algorithm 2. MergeEvent

Input: Similarity threshold θ, Event $E1$, Event $E2$
Output: Events set S

1 **for** $E \in \{E1, E2\}$ **do**
2 $C \leftarrow$ GetSortedMergeCandidate(E);
3 **for** $i \leftarrow 1$ ***to*** $|C|$ **do**
4 $E^* \leftarrow$ Merge(E,$C[i]$);
5 **if** $r_{E^*} < \theta$ **then**
6 S.add(E);
7 break ;
8 **else**
9 S.remove(E);
10 S.remove($C[i]$);
11 $E \leftarrow E^*$;
12 **end**
13 **if** $i == |C|$ **then**
14 S.add(E);
15 **end**
16 **end**
17 **end**
18 **return** S;

Instead of just defining the four event operations, we further work on how to support each of the four operations efficiently. The details are introduced in the following section.

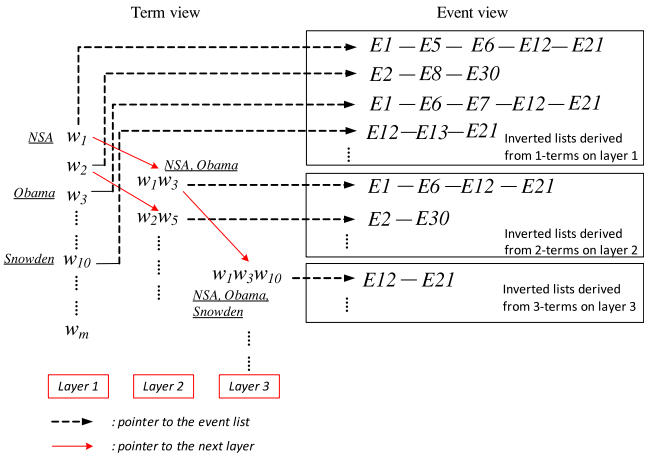


Fig. 1. The multi-layer inverted list structure.

5 Event Indexing and Search

One key process in the incremental event evolution monitoring is to find the nearest neighbour of a new tweet (in the case of create and absorb) or an event (in the case of merge), which is basically a nearest neighbour search. Because tweets are coming in a highly dynamic stream fashion, it is essential to implement the nearest neighbour search efficiently. In this section, we present the event indexing structure named Multi-layer Inverted List which is proposed to index the social events data and support efficient nearest neighbour search for these events. MIL is distinct from the traditional inverted list in several ways.

The inverted file is recognized as the most efficient index structure for text similarity search [20]. As a document is represented as a bag of words, the inverted file maps the words to the documents that contain them. Given a number of query words, this structure highly reduces the search space by limiting the query processing being conducted to the documents which contain at least one of the query words. For event indexing, with the arrival of a new tweet, we aim to find its closest event in order to check if it belongs to any existing event. It is essentially a nearest neighbour search. Thus, a straightforward way to organize events for search is to build an inverted file on them. However, given the fact that tweets are arriving so quickly nowadays, a huge number of events may be generated and updated frequently. The conventional inverted file is no longer sufficient to support large-scale dynamic event management because the lists returned by the inverted file may be still too long. Thus, a more efficient and scalable index structure is necessary. The proposed Multi-layer Inverted List structure organizes events in different layers guided by different information-specific levels, which provides highly efficient search performance for large-scale event sets.

5.1 Index Construction

Different from the conventional inverted file which usually creates entries for individual words, the proposed Multi-layer Inverted List consists of multiple layers, where each entry on the mth layer corresponds to a set of m words referred to as an m-term. An m-term entry points to a list of events, each of which contains all words in this m-term. In more detail, as illustrated in Fig. 1, we build up the entries for 1-terms on layer 1, the entries for 2-terms on layer 2, and so on. More importantly, an $(m+1)$-term is generated by adding a strongly correlated word to its m-term so that the mth layer naturally points to the $(m+1)$th layer. For example in Fig. 1, w_1 stands for "NSA", a 1-term on layer 1. A 2-term w_1w_3 representing "NSA, Obama" is generated and pointed from w_1, and a 3-term $w_1w_3w_{10}$ representing "NSA, Obama, Snowden" is generated and pointed from w_1w_3. The terms on each layer are sorted in alphabetical order. Each m-term entry in this index structure points to a list of events containing the corresponding m words. For instance, the events $E1$, $E6$, $E12$ and $E21$ shown in Fig. 1 all contain the words "NSA" and "Obama". One important feature of MIL is that the event list pointed by an $(m+1)$-term is a subset of the event list pointed by its corresponding m-term. So the events from a shorter list stored on the lower layer (bigger m, contains more words in the index terms) are more likely to be relevant to the new tweet (query tweet) because they share more of the same words. This feature motivates our proposed pruning based search strategy which enables fast search by first inspecting most relevant and shorter event lists on the lowest layer, so as to get a larger similarity value to avoid exhaustive accesses to longer event lists on the upper layers, as will be detailed in Section 5.3. Obviously, the most upper layer (i.e., $m = 1$) of MIL alone can be regarded as the conventional inverted file.

5.1.1 m-Term Generation

The next question is how to generate m-terms on different levels in the proposed indexing structure. We create m-terms according to the vocabulary which is constructed based on the words contained in the collected tweets and which grows dynamically as new tweets arrive. Clearly, not all the words should be selected into the vocabulary, as many of them are trivial and meaningless even after removing the stop words. Thus, how to determine a meaningful word vocabulary is the first step. An intuitive way to measure the significance of words is using their tf·idf values. However, if the tf·idf value of a word is stable over time, most likely it does not indicate the occurrence of events. Emergence of events is usually reflected by the change of words' usage [17]. Given this observation, wavelet analysis [17] is applied in our work to capture the change of the words' usage in tweets. The comparison between the tf·idf based algorithm and the wavelet analysis based algorithm is presented in the experiments.

Essentially, for each word w, a signal S_w can be generated by observing its df·idf values in a certain time period, denoted as:

$$S_w = < s_{w,t_1}, s_{w,t_2}, \ldots, s_{w,t_k} >$$

where $\{t_i | i = 1..k\}$ are a sequence of observation time points within the time window $[t_1, t_k]$ and s_{w,t_i} is a df·idf value of the word w at the time point t_i, which is defined as

$$s_{w,t_i} = \frac{|e \in T_{t_i} : w \in e|}{|e \in T_{t_i}|} \times log \frac{\sum_{j=1}^{i} |T_{t_j}|}{\sum_{j=1}^{i} |e \in T_{t_j} : w \in e|} \quad (2)$$

where T_{t_i} is the collection of tweets with timestamps in the time duration $(t_{i-1}, t_i]$. Notably, df is used here because

multiple appearances of the same word in one short tweet usually indicate the occurrence of the same event [17].

Referring to [17], based on S_w we generate a new signal S'_w to capture the entropy change of the original signal. It is defined as

$$S'_w = < s'_{w,t_1}, s'_{w,t_2}, \ldots, s'_{w,t_k} >$$

where

$$s'_{w,t_i} = \begin{cases} \frac{H_{t_i} - H_{t_i-1}}{H_{t_i-1}}, & \text{if } H_{t_i} > H_{t_i-1} \\ 0, & \text{otherwise.} \end{cases}$$

H denotes the H-Measure of signal S, which is a normalized value of Shannon wavelet entropy. More details can be found from [17]. By calculating the auto-correlation value of S'_w, we can determine whether S'_w is a trivial signal. If the usage of the word w is stable over time, its associated signal S'_w is flat, resulting in a low auto-correlation. S'_w and w will be considered as trivial. In this way, we create the vocabulary by removing the trivial words. Every word in the vocabulary forms a 1-term. For each new word in the arriving tweets, its signal is monitored over the time duration and added into the vocabulary if its auto-correlation is high.

For $m > 1$, m-terms are generated by finding the m-sized frequent word sets from the 1-terms periodically. One option is to re-construct m-terms whenever a significant increase in the vocabulary size is detected. A widely used method FP-growth [31] is applied to achieve efficient and scalable frequent itemset mining. The support threshold λ for FP-growth is tuned in Section 6.2.

The proposed indexing structure consists of m layers. Since each individual tweet is generally a small piece of text consisting of 2-10 meaningful keywords, the length of the largest frequent itemsets is usually small. m is normally in the range of 2-3, as indicated from our experimental results. The effect of m with various values on indexing performance will be further discussed in the experiments.

5.1.2 Event List

Each m-term points to a list of events, where event ids are stored in the list. Some events can contain multiple m-terms. For example, $E12$ (in Fig. 1) is on the lists derived from w_1, w_3, w_{10}, w_1w_3, w_1w_{10}, w_3w_{10} and $w_1w_3w_{10}$ as it contains all these words. It is expected that the lengths of lists on the $(m+1)$th layer are far shorter than those on the mth layer.

Given a tweet e, to identify the event it belongs to, we need to calculate the similarity between e and each individual event. It is critical to reduce the search space as much as we can to support efficient identification. Basically, we only consider event lists from m-term entries, where e contains the corresponding m words. To further reduce the search cost by avoiding exhaustive and expensive similarity computations, we will calculate the similarity upper bound for the remaining events with respect to e. In order to compute upper bounds, the information about each event also needs to be maintained in MIL.

On the mth layer in MIL, for each event E in the list pointed to by an m-term, the necessary information to calculate its upper bound (Equation (3)) with respect to a tweet is the sum of the largest m values and the $(m+1)$th largest value among all dimensions in E. Denoting the ith largest value in E as $\overline{E_i}$, the additional information stored in the indexing structure at the mth layer for E includes a value-pair $< \sum_{i=1}^{m} \overline{E_i}, \overline{E_{m+1}} >$.

Take the event $E12$ (in Fig. 1) as an example. Its associated value-pair on the entry w_1 is $< \overline{E12_1}, \overline{E12_2} >$, the value-pair on the entry w_1w_3 is $< \overline{E12_1} + \overline{E12_2}, \overline{E12_3} >$ and the value-pair on the entry $w_1w_3w_{10}$ is $< \overline{E12_1} + \overline{E12_2} + \overline{E12_3}, \overline{E12_4} >$.

5.2 Incremental Index Maintenance

Whenever there is an update on events upon the arrival of a new tweet, in addition to the event database, the indexing structure needs to be updated as well. We maintain the update of the index incrementally. Specifically, after any of the four event evolution operations happens, the index will be correspondingly updated by the new or updated events. For an MIL structure with m layers ($m \leq 3$ as discussed in Section 6.2), the time complexity of the index maintenance is $O(N^m)$, where N is the number of words in the updated event. In Algorithm 1, the update of MIL will happen after line 2, line 9 and line 15. The new (or updated) events will be inserted into the MIL structure at different layers by matching the terms in MIL. Note that the vocabulary (i.e., the 1-terms) is incrementally enlarged by including the new non-stop-words in the new events, and m-terms at lower layers are periodically generated from tweets. It is important to note that split or merged events are regarded as new events which are inserted into the database and indexed by MIL. The old events (events before split or merge) will be dropped from MIL (line 14 in Algorithm 1 and line 9-10 in Algorithm 2). For example, if $E1$ is split into $E2$ and $E3$, $E1$ will be removed from MIL while $E2$ and $E3$ will be inserted into MIL. In this way, at any point in time, only the active events at that moment are searchable through MIL.

Next, we present the algorithms for nearest neighbour search (i.e., line 5 in Algorithm 1) through MIL. The detailed upper bound estimation based on the information stored in MIL for the search scenario is explained.

5.3 Nearest Neighbour Search

For a new arriving tweet or newly split event, we are first interested in identifying its most similar event from the existing events indexed by MIL, corresponding to nearest neighbour search. To reduce the computational cost, it is critical to design an effective similarity upper bound for quick candidate pruning to avoid full similarity computations. In this section, we will discuss the upper bound estimation and the corresponding search strategy for nearest neighbour search, for a newly arriving tweet e. The procedure for a newly split event is the same.

5.3.1 Upper Bound Calculation

In the proposed MIL indexing structure, an event E may appear in multiple event lists across different layers. As we mentioned before, the events lists on the lower layer are shorter and more likely to be relevant to the given tweet e. Hence, the idea here is to associate the event with different similarity upper bounds at different layers for e. Such a pruning based search strategy along with the proposed multi-

layer structure enable efficient search by first inspecting most relevant event lists on the lowest layer, to avoid exhaustive accesses to longer event lists on the upper layers.

We adopt the Cosine similarity to measure the similarity between an event E and a tweet e, denoted as $\varphi(e, E)$. Notably, in this work, all the vectors (both e and E) are normalized. So the similarity between e and E is calculated as $\varphi(e, E) = \sum_{k=1}^{n} e_k \times E_k$, where e_k denotes the kth dimension in e. The estimation of upper bound of the Cosine similarity $\varphi(e, E)$ for nearest neighbour search is described in the following lemma.

Lemma 1 (Upper Bound). *Given a tweet e and an event E on the event list at the mth layer, the following similarity, denoted as $\varphi_u^m(e, E)$, is an upper bound of $\varphi(e, E)$:*

$$\varphi_u^m(e, E) = \left(\sum_{i=1}^{m} \overline{E_i} + \overline{E_{m+1}} \times (Z - m) \right) \times \overline{e_1} \qquad (3)$$

where $\overline{E_i}$ is the ith largest value among all dimensions of E, $\overline{e_1}$ is the largest value among all dimensions in e, and Z is the number of dimensions which have non-zero values in both E and e.

Proof. When the similarity calculation is required between e and E on the list pointed to by an m-term, it indicates that e and E have at least m common words, which correspond to m dimensions. By assuming that the values of these m dimensions in E are the largest m values among all dimensions, the similarity contributed by them is no greater than

$$\sum_{i=1}^{m} \overline{E_i} \times \overline{e_1} \qquad (4)$$

where $\overline{e_1}$ is the largest value among all dimensions in e.

Besides these m words, E and e may still share some other common words. Denote the total number of common words shared by e and E as Z. It is just the number of dimensions which have non-zero values in both E and e. The similarity contribution from the other $(Z - m)$ common words is no greater than

$$\overline{E_{m+1}} \times \overline{e_1} \times (Z - m). \qquad (5)$$

Taking into account the above two parts, we derive that $\varphi_u^m(e, E) \geq \varphi(e, E)$. □

When m increases, the upper bound $\varphi_u^m(e, E)$ decreases monotonically for the event E with respect to e. In other words, a tighter upper bound for E is provided for a larger m.

Given that m is generally far smaller than the number of words in tweets, the upper bound computation in Equation (3) (with time complexity $O(1)$) is expected to be more efficient than the Cosine similarity computation which has a time complexity linear to the number of words in the tweet. This has been verified in our experiments (Fig. 8). Similar to threshold algorithm (TA) [32], our upper bound is calculated during the process of nearest neighbour search. However, TA needs to maintain multiple sorted lists of objects. For highly dynamic tweets data, frequently updating these sorted lists leads to non-ignorable extra time cost especially for the long lists. Next, we discuss the search strategy for a new tweet to find the most similar event from MIL, by utilizing the upper bound established above.

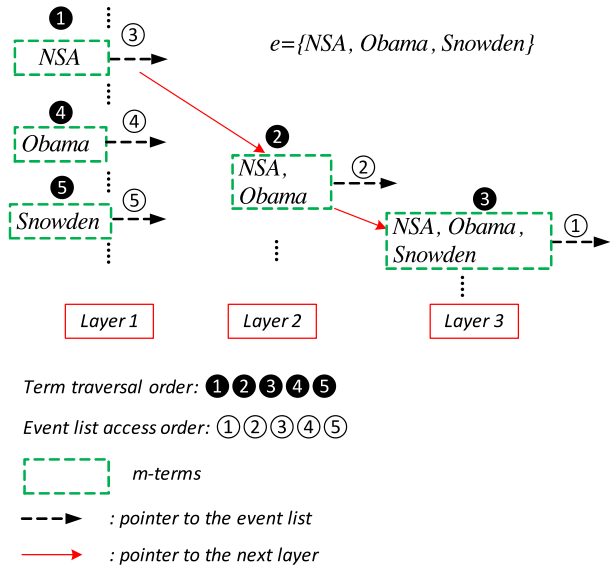


Fig. 2. An example for the term traversal order and the event list access order in MIL.

5.3.2 Search Strategy

The search against the proposed indexing structure MIL is to find the most similar event E from all existing events for a new arriving tweet e according to the Cosine similarity measure (i.e., the NearestNeighbourSearch method at line 5 in Algorithm 1). Given a query, the events at the $(m+1)$th layer in MIL are not guaranteed to have higher similarities to the query than those at the mth layer. However, it is more likely that higher similarities can be obtained at lower layers because the more dimensions/words shared by the query and an event, the more likely the Cosine similarity is large. Hence, by searching the terms with larger m values first, high similarities are more likely to be obtained in short time since events lists at lower layers are much shorter. The biggest advantage of doing so is to approach the most similar event's true similarity as soon as possible so that more events can be quickly pruned based on their established upper bounds to the query. If an event's upper bound is equal to or smaller than the currently found maximum event similarity to the query, it can be safely pruned without computing its actual Cosine similarity.

The general idea is to perform depth-first traversal in MIL and filter the events based on upper bound pruning. Let us elaborate it with our running example, as in Fig. 2 where a partial 3-layer MIL is depicted. Given a new tweet e containing three keywords "NSA", "Obama" and "Snowden", we first locate the lowest layer that e can reach in the indexing structure, by traversing the terms starting with "NSA". In this example, it can reach the third layer with a 3-term containing all three words. The event list pointed to by the 3-term is then accessed. Next, the event list pointed to by the 2-term containing "NSA" and "Obama" is accessed, and followed by the three 1-terms' lists to be accessed. In Fig. 2, the term traversal order and the event list access order are shown in black and blank circles, corresponding to the depth-first pre-order and the depth-first post-order traversal, respectively.

Multiple event lists at each layer can be combined into a single list. Assuming m layers have been traversed in MIL,

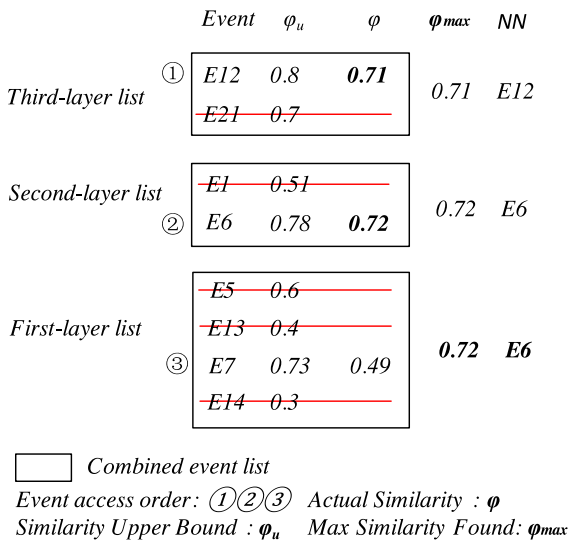


Fig. 3. An example for upper bound pruning.

we can have m combined lists. The events belonging to more than one layer should only be retained in the list at their lowest layer (largest m) where the tightest upper bounds for events are computed. Meanwhile, the same event at the same layer is recorded only once in the combined list. For the example shown in Fig. 1 and for the query tweet e containing three words "NSA", "Obama" and "Snowden", $E12$ and $E21$ only appear in the third-layer combined list, $E1$ and $E6$ are recorded in the second-layer combined list, and $E5$, $E7$, $E13$, and $E14$ are stored in the first-layer combined list.

After we get the combined event lists, the next step is to compute their upper bounds φ_u^m for the query, based on which effective pruning of events can be performed. By avoiding a large number of expensive Cosine similarity computations, the nearest neighbour, i.e., the most similar event, can be found quickly. The combined list at the lowest level is processed first, followed by its next upper level until the most upper level. The largest similarity value found so far, denoted as $\varphi_{\max}$, is maintained during the process. If the upper bound of an event is greater than $\varphi_{\max}$, the event is accessed and the actual Cosine similarity is calculated to check whether the nearest neighbour and $\varphi_{\max}$ need to be updated. Otherwise, it can be safely pruned since its similarity upper bound is already equal to or smaller than $\varphi_{\max}$.

An example for upper bound pruning is illustrated in Fig. 3, where the third-layer to the first-layer combined lists are {$E12$,$E21$}, {$E1$,$E6$}, and {$E5$,$E7$,$E13$,$E14$} respectively. Their upper bounds φ_u are also indicated. The third-layer list is first processed. After the event $E12$ is accessed, $\varphi_{\max}$ and the nearest neighbour are updated to be 0.71 and $E12$. Since the upper bound of $E21$ is smaller than current $\varphi_{\max}$, $E21$ can be pruned directly without computing its Cosine similarity. Then the second-layer list is processed. Since $E1$ has a smaller upper bound than $\varphi_{\max}$, it is pruned. $E6$ is accessed for Cosine similarity computation since its upper bound is greater than $\varphi_{\max}$. The actual similarity of $E6$ is 0.72, which is greater than $\varphi_{\max}$. Therefore, $\varphi_{\max}$ and the nearest neighbour are updated to be 0.72 and $E6$ respectively. Finally, the first-layer list is processed, and only $E7$ is accessed. $E6$ remains as the final result for the nearest neighbour. In this example, only three events ($E12$, $E6$ and $E7$) are required to calculate the actual similarities to the query based on upper bound pruning, with an additional cost on upper bound computations.

Notably, the upper bound calculation requires the known value of Z for each event. In our implementation, give a query, Z can be simply calculated by retrieving all its 1-term event lists and counting the number of occurrences across all the lists for each event. The search algorithm is outlined in Algorithm 3.

In Algorithm 3, by traversing the indexing structure MIL, the maximum number of layers reached by the query tweet e, i.e., m, and the combined list at each layer are first generated (line 1), followed by computing Z value for each event which shares at least one common word with e (line 2). To find the nearest neighbour to e, φ_{max} is initialized to be 0 (line 3). The combined event lists are then accessed in the bottom-up order (lines 4-15), given that events at lower layers potentially have higher similarities than those at upper layers. For each event in the ith layer list, its upper bound φ_u^i is computed (line 6). If $\varphi_u^i > \varphi_{max}$, its actual similarity to the query is computed and compared with φ_{max} to see whether the currently maintained nearest neighbour and its similarity to the query need to be updated (lines 7-13). Otherwise, the event is filtered without similarity computation.

Algorithm 3. Nearest Neighbour Search

```
   Input: Index I, Tweet e
   Output: Nearest Neighbour Event E_NN
 1 Lists[m] ← GenerateLayeredLists(e, I);
 2 Z[] ← ComputeZ(e, I);
 3 φ_max ← 0;
 4 for i ← m to 1 do
 5     for E ∈ Lists[i] do
 6         φ_u^i(e, E) ← ComputeUpperBound(e, E, Z);
 7         if φ_u^i(e, E) > φ_max then
 8             φ ← ComputeCosineSimilarity(e, E);
 9             if φ > φ_max then
10                 φ_max ← φ ;
11                 E_NN ← E ;
12             end
13         end
14     end
15 end
16 return E_NN;
```

The time complexity of the algorithm is linear to the number of events that share at least one common word with the query. However, the proposed upper bound pruning strategy starting from lower to upper layers can greatly reduce the search cost by saving a large number of the Cosine similarity computations. As the query tweet's length increases, such reduction becomes more significant since the Cosine similarity computation gets more expensive. This will be further verified by our experiment results.

6 Experiments

In this section, we report the results of an extensive performance study conducted on a large real-life tweet dataset.

The experiments are designed to verify the effectiveness of the event evolution monitoring process and the efficiency of the index structure.

6.1 Set Up

6.1.1 Dataset

We collected 11,121,112 tweets posted from July 25th, 2013 to November 30th, 2013 using Twitter Search API. There are two types of APIs provided by Twitter to crawl tweets, Stream API and Search API. The former one can only return a very small fraction of the total volume of tweets at any given moment, based on which events can rarely be detected and monitored. Thus, we use Search API to crawl tweets by giving a list of search terms to get more "meaningful" tweets. We construct the search terms list by including a set of event-relevant words (e.g., accident, hurricane, etc.), as well as the top 10 hottest trend words from the G20 countries in every two hours. The content of each tweet is preprocessed by removing stop words, stemming, and obtaining nouns and verbs only. The tweets are incrementally processed in the order of their posted time to simulate the dynamic tweets stream, and the generated events can be reported at any time. To evaluate the performance of the proposed indexing method, we generate four subsets in sizes of 2 million, 4 million, 6 million and 8 million tweets in terms of their posted time. 100,000 tweets are randomly picked as new coming tweets (i.e., queries), with the length ranging from 2 to 10 words, to test the effect of query length on the efficiency of nearest neighbour search. Note that the data volume is relatively small on Twitter scale, this is because the tweets that can be crawled by Twitter API are limited and we further restrict the collected tweets to specified countries and search terms. Considering that 58 million tweets are posted everyday on average, we will extrapolate our results to the entire tweet stream in a sliding window of two weeks later.

6.1.2 Ground Truth Generation

Due to the lack of ground truth for event detection and evolution monitoring from Twitter, the evaluation metrics for TDT (Topic Detection and Tracking) cannot be directly used for our work. We first build up the ground truth (the top-N hottest events that happened on a particular day) from the tweet stream. In our data collection process, each tweet is collected based on a search term. Given a set of daily collected tweets, the sum of re-tweeted numbers is calculated for every search term as the mentioned times. We pick a list of search terms with the top-N largest mentioned times as the event ground truth for that day, referred to as ground-truth-N. It is not practical to construct ground truth for event evolution monitoring. Thus a user study is applied in the evaluation phase.

6.1.3 Performance Indicators

Three metrics are adopted to evaluate the effectiveness of the process of monitoring evolving events.

- Purity: The purity is defined as the percentage of labelled objects of the majority class in each cluster for all the clusters. As a standard measure of the cluster quality, it can be used to check the quality of an event which is basically a cluster of tweets.
- Precision: Precision is a classical and effective measure in information retrieval to evaluate the accuracy. In this paper, it is calculated by dividing the number of events existing in both retrieved top-N events from our algorithm and the corresponding ground-truth-N by N.
- Coverage: Coverage is the proportion of the total ground truth events that have been detected. Specifically, it is calculated by dividing the number of different events existing in both retrieved top-N events from our algorithm and the corresponding ground-truth-N by N. In other words, when similar events correspond to one ground truth event, we will count it once only. The reason we choose coverage in addition to precision is that we want to know how many ground truth events are detected correctly, not how many detected events are ground truth events. The former is more important in real life, since recommending similar events is useless.

Three measures are used to evaluate the performance of the index structure for nearest neighbour search.

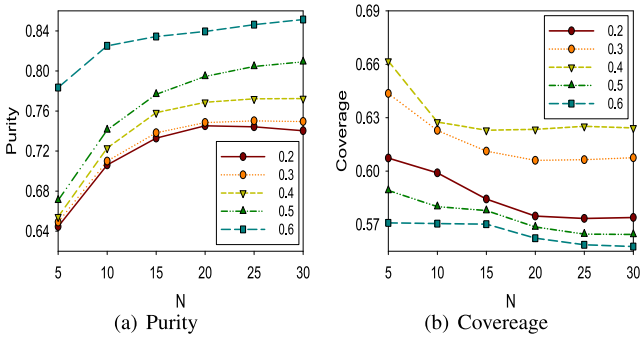
- Pruning power: Pruning power is the percentage of events that are pruned in the filtering step of search processing.
- Search time: Search time is the total query response time for a tweet to get its nearest neighbour event from the index. It dominates the overall time for processing a new tweet.
- Index update time: Index update time is the total time spent on updating the index after an event is updated by a new tweet.

6.1.4 Compared Methods

We design two comparison strategies in the experiments for evaluating the performance of the evolving events monitoring and the index structure respectively.

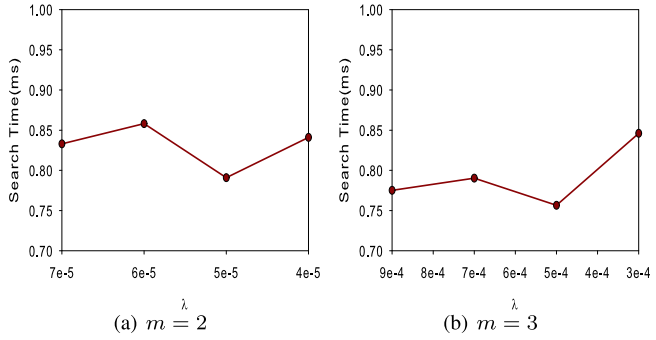
Event evolution monitoring. We propose four operations to monitor evolving events over time. Hence we compare our event evolution monitoring method with the traditional single-pass incremental clustering method to show the effectiveness of the four operations. There are two variants for single-pass incremental clustering algorithm: the one with predefined threshold [3] and the one with automatically adjusted threshold [19], referred to as SPIC1 and SPIC2 respectively. Moreover, to compare with the state-of-the-art event evolution tracking method, eTrack [5] which is based on graph partition algorithm is implemented and compared. To demonstrate the superior 1-terms generated by the wavelet analysis algorithm, the EEM-TFIDF is compared, which is our proposed event evolution method with the 1-terms consisting of the words with high tf·idf values.

Index structure. We compare our Multi-layer Inverted List with three other methods. The first one is the traditional inverted list indexing structure, which is classic and effective for text retrieval. The remaining two methods are IL with different upper bounds for the Cosine similarity. One is the upper bound proposed in [33], referred to as IL_{φ_1}. The other is our proposed upper bound without the multi-layer



(a) Purity (b) Covereage

Fig. 4. Tuning θ_U.



(a) $m = 2$ (b) $m = 3$

Fig. 5. Tuning λ.

structure, referred to as IL_{φ_2}. Notably, the indexing structure in [23] is not compared because their method is proposed to reduce the IO cost which is not appropriate to be tested in a main memory setting.

All experiments are implemented on a desktop with Intel (R) Core (TM) i7-3770 CPU @3.40 GHz processor and 8 GB memory. The operating system is Windows 7 Enterprise (64-bit).

6.2 Tuning of Parameters

Two parameters need to be tuned in our method. They are θ_U for the event evolution monitoring and λ for the MIL index structure. The 439,880 tweets posted from July 25th to July 31st are used for the tuning of parameters and the remaining 10,681,232 tweets are used for testing.

θ_U controls the radius threshold for events to avoid the side effect of extremely large threshold values. The effect of θ_U on purity and coverage for different θ_U values in the range of 0.2 to 0.6 is shown in Fig. 4. In Fig. 4a, the purity increases as θ_U goes up for different top-N hottest events. This is because the higher the threshold is, the less noise will be inserted into the clusters. In Fig. 4b, the coverage for event detection first increases when the threshold rises. It reaches its peak when $\theta_U = 0.4$ and then goes down. The reason is that when large events are split into small events due to the high threshold, the top few largest events cannot be correctly tracked anymore. In consideration of both coverage and purity, we set 0.4 as the default value for θ_U.

λ is the support threshold for FP-growth which is used to control the generation of m-terms at each layer in MIL. The 1-terms are generated by wavelet analysis based on the goal to make more than 99 percent of tweets contain at least two words (after removing stop-words). This is because tweets containing just one word may lead to ambiguity or misunderstanding about its topic. From the 1-terms, we generate 2-terms by tuning λ to get the best performance of the 2-layer MIL. Similar idea is applied to tune λ for different layers in MIL. Fig. 5 shows the results for $m = 2$ and 3 respectively, from which we set $\lambda = 0.00005$ and 0.0005 to generate 2-terms and 3-terms respectively.

θ_U and λ are used to generate events and terms. Table 3 shows the statistics for five tweet datasets, where Avg. Size means the average number of tweets in each event, Avg. Length means the average length of the event lists at each layer of MIL, and No. of Terms means the number of terms at each layer. We do not show the results for $m > 3$, as their numbers of terms generated are very small and most tweets cannot reach those extremely low layers. As we can see, we have less than 20,000 1-terms (i.e., the vocabulary size) for the largest dataset. The average event list length for the first layer is mostly around 100. For the second layer, the number of 2-terms increases dramatically, while the average list length drops quickly to less than 24 even for the largest dataset. This is reasonable since there are many tweets that could share two common words. However, the possibility for tweets to share more than two words plunges, as indicated by the number of 3-terms in the table, and the average list length is further reduced to be around 10 or slightly more.

Comparing the index size of MIL with the traditional IL, i.e., the first layer of MIL, MIL consumes much larger space than IL. For the largest dataset, IL occupies about 8 MB space, while MIL takes about 80 MB. However, compared with the main memory size of 8 GB, such space cost is rather small. Therefore, we assume all events and the index structure are fully maintained in main memory for the experiments. Despite the disadvantage of space requirement for MIL, we focus on its search performance improvements.

From the space consumption of different sizes of datasets in our experiments, we can extrapolate that for the entire tweet stream in a sliding window of two weeks (812 million tweets), MIL takes about 6.2 GB space. And quite a large percentage of events (e.g., 62 percent in our datasets) only contain one tweet. These events are created by the noisy

TABLE 3
Data Statistics

			m = 1		m = 2		m = 3	
No. of Tweets	No. of Events	Avg. Size	Avg. Length	No. of Terms	Avg. Length	No. of Terms	Avg. Length	No. of Terms
2 million	88,723	12.82	68.86	8,798	13.51	189,721	9.56	11,858
4 million	121,451	14.05	78.42	11,019	15.73	332,806	10.76	26,032
6 million	165,260	14.58	89.79	13,625	18.79	475,226	13.40	42,829
8 million	203,608	14.74	97.98	15,693	20.52	625,470	14.43	57,099
10 million	260,925	14.78	109.78	18,275	23.56	774,864	15.31	66,225

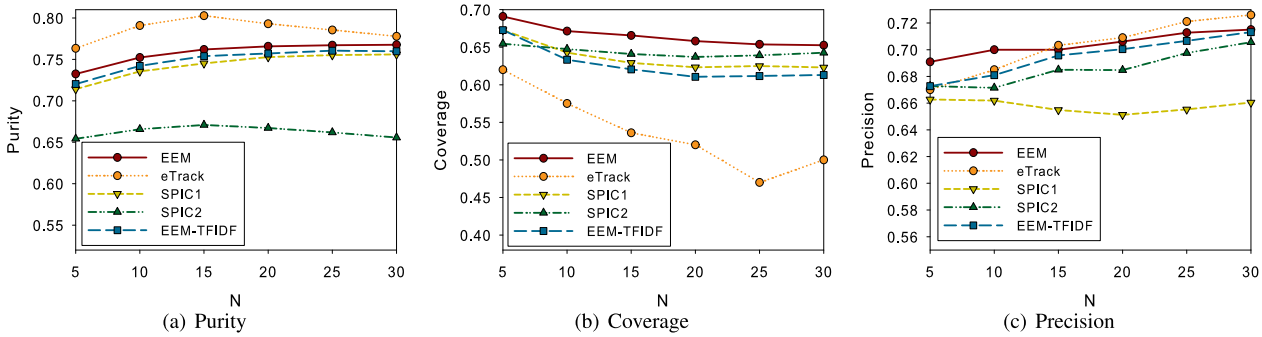


Fig. 6. Event detection performance.

tweets which are not relevant to any event. By filtering out these singleton events out of the sliding window, the space consumption will be further reduced. Hence our method is capable of processing the entire tweet stream in reality with a sliding window of a fortnight under our current computer configurations.

6.3 Results on Event Evolution Monitoring

Our proposed method can detect events from tweets stream and monitor their evolutions. In this experiment, we first test the performance of the event detection and then verify the performance of evolution monitoring. A special dataset named "Tweets-Lite" is used in this experiment, which consists of 500,634 tweets collected from August 1st, 2013 to August 15th, 2013.

6.3.1 Event Detection

In this part, different event detection methods are compared. The coverage and purity are reported for different N values from 5 to 30, where N means the top-N hottest events detected. For each day, the top-N hottest events are monitored and the corresponding purity, precision and coverage are calculated. The average purity, precision and coverage for different N are reported as the final results.

As illustrated in Fig. 6, our method EEM outperforms almost all the others in all the three performance indicators except for purity of eTrack. In general, SPIC1 is better in purity but worse in coverage. This is because the similarity threshold can be adjusted to make the cluster smaller so that the noise in an event is less. In this case, the purity will be high. However, big events will be separated into several similar small events. Hence the coverage will decrease since some biggest events are not accurately captured. EEM adopts the automatic threshold. It is also able to trigger the split or merge operation to have a good balance on event sizes and noises for event evolution monitoring. eTrack shows the highest purity but relatively low coverage. The high purity comes from their defined skeletal cluster which effectively removes the noises in the clusters. However, in their method, multiple recommended events correspond to the same ground truth event leading to the unsatisfactory coverage value. As for the algorithm of filtering words to generate the 1-terms, the wavelet analysis algorithm (EEM) outperforms the term frequency based algorithm (EEM-TFIDF) in all three measurements: purity, precision and coverage. This shows the effectiveness of the wavelet analysis. Moreover, we further check the words removed by wavelet analysis and find that around 3 percent of event-relevant keywords are incorrectly filtered. However, considering that there are 83 percent noisy words removed in total, which significantly saves the space and time consumption, the little sacrifice in accuracy is acceptable. Unlike in coverage, eTrack shows comparable performance to EEM in precision (Fig. 6c). Comparing Figs. 6b and 6c, we can find that around 70 percent events detected by eTrack are ground truth events, however 10-20 percent of them are similar (duplicate). As for EEM, the percentage of duplicate events is under 5 percent.

6.3.2 Evolution Monitoring

In this section, we focus on the performance study on the event evolution monitoring.

We first compare the performance of split and merge operations between the state-of-the-art evolution monitoring method eTrack and our method EEM. We manually check the evolution results of these two methods within the time window of two days. Moreover, SPIC1 is also inspected to further check the influence of split and merge operations on traditional single pass incremental clustering algorithm. Table 4 lists the statistics of event evolution operations. We can see that EEM generates more events than eTrack due to the skeletal cluster designed in eTrack. The Running Time shows the time cost on processing all the tweets within the two-day time window. The proposed EEM outperforms eTrack by reducing more than 70 percent time cost. Comparing the performance of EEM and SPIC1 shown in both Fig. 6 and Table 4, it is obvious that split and merge operations improve the event evolution performance on purity, coverage and precision with only slight additional time cost.

To better understand the event evolution details of our EEM, we also conduct a case study on the dataset "Tweets-Lite" as shown in Table 5 and Fig. 7. "No. of Occurrences"

TABLE 4
Event Evolution Statistics

Methods	No. of Events	No. of Split	No. of Merge	Running Time (ms)
eTrack	459	75	6	285,177
EEM	2,465	74	17	84,970
SPIC1	2,412	-	-	81,095

TABLE 5
Event Evolution Operations Case Study

Factors	Creation	Absorption	Split	Merge
No. of Occurrences	13,792	485,642	1,200	104
Avg. Event Size	-	-	52.08	3.22

is the number of occurrences of each operation. The "Avg. Event Size" is the average size of the events before they are split or merged. From Table 5, we can see that most tweets are absorbed by an existing event. Generally, split operations happen on the large events while the merge operations happen on the small events. Only around 8 percent split operations are followed by a merge operation.

The distribution of the event durations is shown in Fig. 7. We can observe that, 23.1 percent events last just one day, and this number decreases to half when we count the events that last for two days. 17.7 percent events are long-running events which are active all the time. For the other durations, the number of events stays stable.

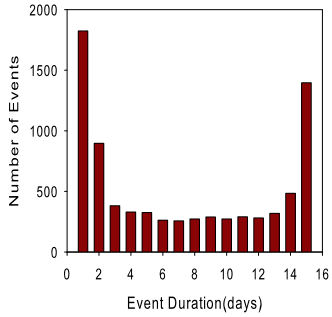


Fig. 7. Distribution of event duration.

6.4 Results on Indexing

In this section, we report the comparison results on the indexing performance for the nearest neighbour search. To show the effect of data size, we conduct experiments on the five datasets as mentioned in Section 6.1.1. Their events and terms are first generated and then indexed by MIL. The effect of the query length is also examined, given the range from 2 to 10.

The search time and the pruning power of different methods with different lengths of query tweets on different sizes of datasets are shown in Fig. 8. Figs. 8a, 8b, 8c, 8d, and 8e show the results of search time for different index methods on different sizes of datasets. Clearly, MIL achieves the best search time and larger datasets lead to more search time for all methods. However, the increments for our method MIL are much smaller than the traditional IL. Equipped with two different upper bounds, IL can be further improved by IL_{φ_1} and IL_{φ_2}. Our proposed upper bound significantly outperforms the upper bound proposed in [33]. By applying the multi-layer structure, the search time can be further improved. As the query length increases, the superiority of MIL is better demonstrated. Figs. 8f, 8g, 8h, 8i, and 8j show the pruning power of IL_{φ_1}, IL_{φ_2}, and MIL. IL does not prune any events from their full similarity computations. All datasets show similar trends. Our proposed

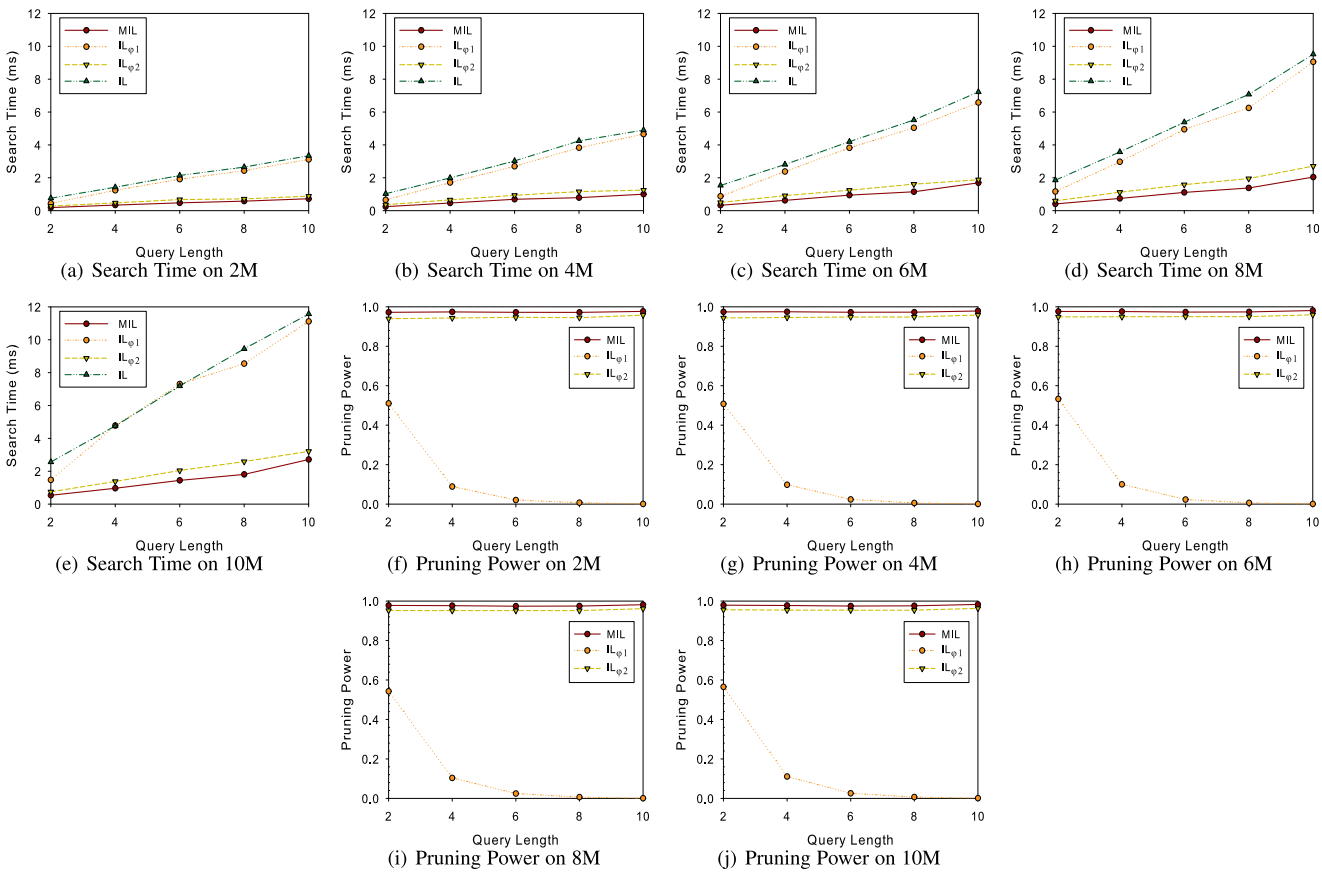


Fig. 8. Effect of query length and data size for nearest neighbour search.

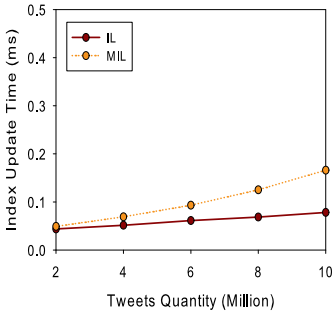


Fig. 9. Effect of data size on index update time.

upper bound is able to prune over 95 percent of the events and together with the multi-layer structure it can further increase the pruning power to be 98 percent for all different query lengths and data sizes, showing the strong robustness of MIL. $IL_{\varphi 1}$ can achieve less than 60 percent pruning power and its capability drops quickly as query length increases. The pruning power is close to zero when the query length reaches 8.

We also test the index update time whenever there is an update on events as described in Section 5.2. As shown in Fig. 9, the index update time is minor compared to the search time. This is because in addition to the time spent on retrieving events list from MIL, search time also includes the time on upper bound calculation and a small number of similarity calculations, while the index update time only includes the time spent on updating events in MIL. As the data size increases, the index update time grows as well. Due to the multi-layer traversal, MIL takes slightly more time than the one layer IL. However, compared with the significant gain in performance of nearest neighbour search, additional cost for index update is acceptable.

We further test the average and maximum index maintenance time after each of the four event operations on dataset "Tweets-Lite" and the results are listed in Table 6. The time cost on the index maintenance depends on the number of words contained in the event that is to be updated in MIL. For creation, the new event contains just one tweet. From Table 5 we can see that the event size is also small for merge. Hence the average index maintenance times for creation and merge are much smaller than for the other two operations. The large number of words in events will lead to a considerable increase in index maintenance time. That's why absorption takes significantly longer time than others to maintain MIL. For the worst case, index maintenance time after creation is the smallest among the four operations because the event to be updated in the indexing structure only contains one tweet (the number of words is limited).

To better understand the four event operations in the proposed framework, a running time analysis is conducted for these operations. Note that for creation and absorption, the processing time basically comes from the nearest neighbour search. Hence in Table 7 the average and maximum processing time of the nearest neighbour search, split and merge are listed. Theoretically, nearest neighbour search, split (bisecting K-means) and merge all have a linear time complexity (linear in the number of the events retrieved from MIL, linear in the number of tweets contained in the split event and linear in the the number of merge candidate events retrieved from MIL respectively). From Table 7 we can see that, on average nearest neighbour search takes the least time, and split takes the most. This is because split (bisecting K-means) makes multiple iterations on each run while the other two operations are just one-pass. In the worst case, merge is the slowest operation. The reason is that, after merging with each single candidate event, the radius of the newly merged event will be calculated to check if the stop condition is met. This will be time-consuming if the merge candidate list is long.

TABLE 6
Index Maintenance Time After Four Event Operations

Case	Creation (ms)	Absorption (ms)	Split (ms)	Merge (ms)
Average	0.003	0.188	0.020	0.007
Worst	0.497	49.580	0.700	0.620

TABLE 7
Running Time Analysis

Case	Nearest Neighbour Search (ms)	Split (ms)	Merge (ms)
Average	1.754	23.857	6.917
Worst	273.732	124.350	468.523

7 Conclusion

In this paper, we have presented a novel event monitoring method along with a multi-layer inverted indexing structure to efficiently and effectively index evolving events from tweet streams. Four operations are designed to capture the dynamics of events over time. A multi-layer structure is proposed to improve the performance of the traditional inverted file and used to accelerate the event evolution monitoring process. Extensive experiments are conducted on real-life tweet datasets to verify the novelty of our methods. In future, we plan to study the effectiveness and efficiency of event monitoring using techniques that incorporate temporal decay [34]. Temporal information is a significant factor for social data like tweets. With the consideration of time, the event detection performance will be further improved in both accuracy and scalability.

References

[1] J. Allan, R. Papka, and V. Lavrenko, "On-line new event detection and tracking," in *Proc. 21st Annu. Int. ACM SIGIR Conf. Res. Develop. Inf. Retrieval*, 1998, pp. 37–45.
[2] H. Becker, M. Naaman, and L. Gravano, "Learning similarity metrics for event identification in social media," in *Proc. 3rd Int. Conf. Web Search Web Data Mining*, 2010, pp. 291–300.
[3] Y. Jie, L. Andrew, C. Mark, R. Bella, and P. Robert, "Using social media to enhance emergency situation awareness," *IEEE Intell. Syst.*, vol. 27, no. 6, pp. 52–59, Nov.-Dec. 2012.
[4] A. Marcus, M. S. Bernstein, O. Badar, D. R. Karger, S. Madden, and R. C. Miller, "Twitinfo: Aggregating and visualizing microblogs for event exploration," in *Proc. SIGCHI Conf. Human Factors Comput. Syst.*, 2011, pp. 227–236.
[5] P. Lee, L. V. S. Lakshmanan, and E. E. Milios, "Incremental cluster evolution tracking from highly dynamic network data," in *Proc. IEEE 30th Int. Conf. Data Eng.*, 2014, pp. 3–14.
[6] A. Gruenheid, X. L. Dong, and D. Srivastava,, "Incremental record linkage," *Proc. VLDB Endowment*, vol. 7, no. 9, pp. 697–708, 2014.
[7] H. Abdelhaq, C. Sengstock, and M. Gertz, "Eventweet: Online localized event detection from twitter," *Proc. VLDB Endowment*, vol. 6, no. 12, pp. 1326–1329, 2013.

[8] T. Sakaki, M. Okazaki, and Y. Matsuo, "Tweet analysis for real-time event detection and earthquake reporting system development," *IEEE Trans. Knowl. Data Eng.*, vol. 25, no. 4, pp. 919–931, Apr. 2013.
[9] C. Li, A. Sun, and A. Datta, "Twevent: Segment-based event detection from tweets," in *Proc. Conf. Inf. Knowl. Manage.*, 2012, pp. 155–164.
[10] R. Li, K. H. Lei, R. Khadiwala, and K. C.-C. Chang, "Tedas: A twitter-based event detection and analysis system," in *Proc. Int. Conf. Data Eng.*, 2012, pp. 1273–1276.
[11] F. Atefeh and W. Khreich, "A survey of techniques for event detection in twitter," *Comput. Intell.*, vol. 31, pp. 132–164, 2015.
[12] H. Gu, X. Xie, Q. Lv, Y. Ruan, and L. Shang, "Etree: Effective and efficient event modeling for real-time online social media networks," in *Proc. IEEE/WIC/ACM Int. Conf. Web Intell. Intell. Agent Technol.*, 2011, pp. 300–307.
[13] R. Lee and K. Sumiya, "Measuring geographical regularities of crowd behaviors for twitter-based geo-social event detection," in *Proc. Int. Workshop Location Based Soc. Netw.*, 2010, pp. 1–10.
[14] K. Massoudi, M. Tsagkias, M. de Rijke, and W. Weerkamp, "Incorporating query expansion and quality indicators in searching microblog posts," in *Proc. 33rd Eur. Conf. Adv. Inf. Retrieval*, 2011, pp. 362–367.
[15] H. Becker, M. Naaman, and L. Gravano, "Beyond trending topics: Real-world event identification on twitter," in *Proc. 5th Int. Conf. Weblogs Soc. Media*, 2011, pp. 438–441.
[16] A.-M. Popescu, M. Pennacchiotti, and D. Paranjpe, "Extracting events and event descriptions from twitter," in *Proc. 20th Int. Conf. World Wide Web*, 2011, pp. 105–106.
[17] J. Weng and B.-S. Lee, "Event detection in twitter," in *Proc. Int. Conf. Weblogs Soc. Media*, 2011, pp. 401–408.
[18] A. Angel, N. Koudas, N. Sarkas, D. Srivastava, M. Svendsen, and S. Tirthapura, "Dense subgraph maintenance under streaming edge weight updates for real-time story identification," *VLDB J.*, vol. 23, no. 2, pp. 175–199, 2014.
[19] C. C. Aggarwal and K. Subbian, "Event detection in social streams," in *Proc. SDM*, 2012, pp. 624–635.
[20] J. Zobel and A. Moffat, "Inverted files for text search engines," *ACM Comput. Surv.*, vol. 38, no. 2, pp. 1–55, 2006.
[21] M. Chang and C. K. Poon, "Efficient phrase querying with common phrase index," in *Proc. 28th Eur. Conf. IR Res.*, 2006, pp. 61–71.
[22] M. Pitts, S. Savvana, S. B. Roy, and V. Mandava, "ALIAS: Author disambiguation in microsoft academic search engine dataset," in *Proc. Int. Conf. Extending Database Technol.*, 2014, pp. 648–651.
[23] M. Terrovitis, S. Passas, P. Vassiliadis, and T. Sellis, "A combination of trie-trees and inverted files for the indexing of set-valued attributes," in *Proc. 15th ACM Int. Conf. Inf. Knowl. Manage.*, 2006, pp. 728–737.
[24] M. Busch, K. Gade, B. Larson, P. Lok, S. Luckenbill, and J. Lin, "Earlybird: Real-time search at twitter," in *Proc. IEEE 28th Int. Conf. Data Eng.*, 2012, pp. 1360–1369.
[25] C. Chen, F. Li, B. C. Ooi, and S. Wu, "Ti: An efficient indexing mechanism for real-time search on tweets," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2011, pp. 649–660.
[26] X. Zhou and L. Chen, "Event detection over twitter social media streams," *VLDB J.*, vol. 23, no. 3, pp. 381–400, Jun. 2014.
[27] R. McCreadie, C. Macdonald, I. Ounis, M. Osborne, and S. Petrovic, "Scalable distributed event detection for twitter," in *Proc. Int. Conf. Big Data*, 2013, pp. 543–549.
[28] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge, U.K.: Cambridge Univ. Press, 2008.
[29] S. Unankard, X. Li, and M. Sharaf, "Emerging event detection in social networks with location sensitivity," *World Wide Web*, pp. 1–25, 2014.
[30] M. Steinbach, G. Karypis, and V. Kumar, "A comparison of document clustering techniques," in *Proc. KDD Workshop Text Mining*, 2000, pp. 1–20.
[31] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," in *Proc. SIGMOD*, 2000, pp. 1–12.
[32] R. Fagin, A. Lotem, and M. Naor, "Optimal aggregation algorithms for middleware," in *Proc. Symp. Principles Database Syst.*, 2001, pp. 102–113.
[33] A. C. Awekar and N. F. Samatova, "Fast matching for all pairs similarity search," in *Proc. IEEE/WIC/ACM Int. Joint Conf. Web Intell. Intell. Agent Technol.*, 2009, pp. 295–300.
[34] E. Cohen, "Decay models," in *Encyclopedia of Database Systems*. New York, NY, USA: Springer 2009, pp. 757–761.

Hongyun Cai received the bachelor's degree in computer science from Nankai University, China, in 2010 and the master's degree in science from the University of Queensland in 2011. She is currently working toward the PhD degree in the School of ITEE at the University of Queensland. Her research interest includes multimedia data mining and social data management and analysis.

Zi Huang received the BSc degree from the Department of Computer Science, Tsinghua University, Beijing, China, and the PhD degree in computer science from the School of ITEE, The University of Queensland, Brisbane, QLD, Australia. She is currently an ARC future fellow in the School of ITEE, The University of Queensland. Her research interests mainly include multimedia indexing and search, social data analysis, and knowledge discovery.

Divesh Srivastava received the BTech degree from the Indian Institute of Technology, Bombay, and the PhD degree from the University of Wisconsin, Madison. He is the head of the Database Research Department at AT&T Labs-Research. He has served as the associate editor-in-chief of the *IEEE Transactions on Knowledge and Data Engineering*, and the program committee co-chair of many conferences, including VLDB 2007. He has presented keynote talks at several conferences, including VLDB 2010. His research interests span a variety of topics in data management. He is a fellow of the ACM, on the board of trustees of the VLDB Endowment, and an associate editor of the *ACM Transactions on Database Systems*.

Qing Zhang is a research scientist at the Australian e-Health Research Centre. He now works on the Health Stream Data Analytics project. During his postdoc, he worked on the Health Data Integration project. From 2004 to 2005, he was engaged in the Streaming Data Processing project and Skyline Query Processing project while working as a research fellow at the University of New South Wales (UNSW). Before that, he worked as a computer engineer at the Institute of Computing Technology of Chinese Academy of Sciences.