Incremental Cluster Evolution Tracking from Highly Dynamic Network Data

Pei Lee #1, Laks V.S. Lakshmanan #2, Evangelos E. Milios *3

Computer Science Department, University of British Columbia, Vancouver, BC, Canada
1,2 {peil,laks}@cs.ubc.ca

* Computer Science Department, Dalhousie University, Halifax, NS, Canada
3 eem@cs.dal.ca

***Abstract*—Dynamic networks are commonly found in the current web age. In scenarios like social networks and social media, dynamic networks are noisy, are of large-scale and evolve quickly. In this paper, we focus on the cluster evolution tracking problem on highly dynamic networks, with clear application to event evolution tracking. There are several previous works on data stream clustering using a node-by-node approach for maintaining clusters. However, handling of bulk updates, i.e., a subgraph at a time, is critical for achieving acceptable performance over very large highly dynamic networks. We propose a subgraph-by-subgraph incremental tracking framework for cluster evolution in this paper. To effectively illustrate the techniques in our framework, we take the event evolution tracking task in social streams as an application, where a social stream and an event are modeled as a dynamic post network and a dynamic cluster respectively. By monitoring through a fading time window, we introduce a skeletal graph to summarize the information in the dynamic network, and formalize cluster evolution patterns using a group of primitive evolution operations and their algebra. Two incremental computation algorithms are developed to maintain clusters and track evolution patterns as time rolls on and the network evolves. Our detailed experimental evaluation on large Twitter datasets demonstrates that our framework can effectively track the complete set of cluster evolution patterns in the whole life cycle from highly dynamic networks on the fly.**

I. Introduction

People easily feel overwhelmed by the information deluge coming from social connections which flow in from channels like Twitter, Facebook/LinkedIn, forums, blog websites and email-lists. There is thus an urgent need to provide users with tools which can automatically extract and summarize significant information from highly dynamic social streams, e.g., report emerging bursty events, or track the evolution of one or more specific events in a given time span. There are many previous studies [1], [2], [3], [4], [5], [6] on detecting new emerging events from text streams; they serve the need for answering the query "*what's happening?*" over social streams. However, in many scenarios, users may want to know more details about an event and may like to issue advanced queries like "*how're things going?*". For example, for the event "SOPA (Stop Online Piracy Act) protest" happening in January 2012, existing event detection approaches can discover bursty activities at each moment, but cannot answer queries like "how SOPA protest has evolved in the past few days?". An ideal output to such an evolution query would be a "panoramic view" of the event history, which improves user experience. In this paper, we model social streams as dynamically evolving post networks and model events as clusters over these networks, obtained by means of a clustering approach that is robust to the large amount of noise present in social streams. Accordingly, we consider the above kind of queries as an instance of the cluster evolution tracking problem, which aims to track the cluster evolution patterns at each moment from such dynamic networks. Typical cluster evolution patterns include birth, death, growth, decay, merge and split. Event *detection* can be viewed as a subproblem of cluster *evolution tracking* in social streams.

There are several major challenges in cluster evolution tracking. In many scenarios, dynamic networks are of large scale and evolve quickly. Thus, the first challenge is the effective design of incremental computation. The traditional approaches [7], [8] based on decomposing a dynamic network into snapshots and processing each snapshot independently from scratch are prohibitively expensive. An efficient single-pass incremental computation framework is essential for cluster evolution tracking over social streams that exhibit very large throughput rates. To our knowledge, surprisingly this not yet been studied. The second challenge is the formalization and tracking of cluster evolution operations under an incremental computation framework, as the network evolves. Most related work reports cluster activity by volume over the time dimension [2], [1]. While certainly useful, this is just not capable of showing the evolution behaviors about how clusters are split or merged, for instance. The third challenge is handling of bulk updates. Since dynamic networks may change rapidly, a node-by-node approach to incremental updating will lead to poor performance. A subgraph-by-subgraph approach to incremental updating is critical to achieve good performance over very large, fast-evolving dynamic networks such as post networks. But this in turn brings new challenge to incremental cluster maintenance against bulk updates.

To handle the above challenges, we propose an incremental tracking framework for cluster evolution over highly dynamic networks. To illustrate the techniques in this framework, we take the *event evolution tracking* task in social streams as an application, where a social stream and an event are modeled as a dynamic post network and a post cluster respectively.

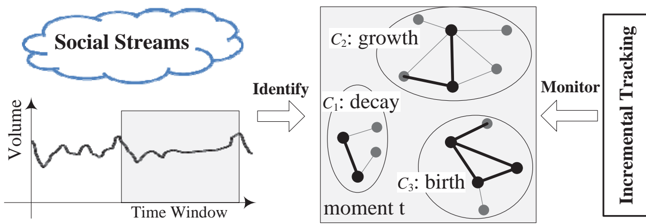


Fig. 1. Post network captures the correlation between posts in the time window at each moment, and evolves as time rolls on. The skeletal graph is shown in bold. From moment t to $t+1$, the incremental tracking framework will maintain clusters and monitor the evolution patterns on the fly.

$S_F(p_1, p_2)$	the fading similarity between posts p_1 and p_2
(ε, δ)	similarity threshold, priority threshold
$w^t(p)$	the priority of post p at moment t
$G_t(V_t, E_t)$	the post network at moment t
G_{old}	the old subgraph that lapses at moment $t+1$
G_{new}	the new subgraph that appears at moment $t+1$
$\overline{G}_t(\overline{V}_t, \overline{E}_t)$	the skeletal graph at moment t
$\overline{C}, \overline{S}_t$	a component, a component set in $\overline{G}_t$
C, S_t	a cluster, a cluster set in G_t
$\mathcal{N}(p)$	post p's neighbor set with similarity larger than ε
$N_c(p)$	the cluster set of post p's neighboring core posts

TABLE I
NOTATION.

The reasons we deploy our framework on this application are, social streams usually surge very quickly, making it ideal for the performance evaluation, and events are human-readable, making it convenient to assess the quality. In detail, since a significant portion of social posts like tweets are just noise, we first define a *Skeletal Graph* as a compact summary of the original post network, from which post clusters can be generated. Then, as we will discuss later, we monitor the network updates with a fading time window, and capture the evolution patterns of networks and clusters by a group of primitive evolution operations and their algebra. Moreover, we extend the node-by-node evolution to the subgraph-by-subgraph evolution to boost the performance of evolution tracking of clusters. Figure 1 shows an overview of major modules we use for cluster evolution tracking in social streams.

We notice that at a high level, our method resembles previous work on density-based clustering over streaming data, e.g., DenStream in [9] and cluster maintenance in [10] and [11]. However, there are several major differences with this body of work. First, our approach works on highly dynamic networks and provides users the flexibility in choosing the scope for tracking and monitoring new clusters by means of a fading time window, unlike these works. Second, the existing work can only process the adding of nodes/edges one by one, while our approach can handle adding, deleting and fading of nodes, in bulk mode, i.e., *subgraph by subgraph*. This is an important requirement for dealing with the high throughput rate of dynamic networks. Third, the focus of our approach is tracking and analyzing the cluster evolution dynamics in the whole life cycle. By contrast, the previous works focus on clustering streaming data, which is a sub-task in our problem.

On the application side, comparing with traditional topic tracking approaches, we note that they are usually formulated as a classification problem [12]: when a new story arrives, compare it with topic features in the training set by decision trees or k-NN, etc. [13], and if it matches sufficiently, declare it to be on a topic. Since these approaches assume that topics are *predefined before tracking*, we cannot simply apply them to event evolution tracking in social streams. To compare with existing event detection and tracking approaches [1], [2], [3], [4], our framework has advantages in tracking the whole life cycle and capture composite evolution behaviors such as merging and splitting.

In summary, the problem we study in this paper is captured by the following questions: how to incrementally and efficiently track the evolution behaviors of clusters in large-scale weighted networks, which are noisy and highly dynamic? In this paper, we develop a framework and algorithms to answer these questions. Our main contributions are the following:

- We propose an incremental computation framework for cluster evolution on highly dynamic networks (Sec. III);
- We filter out noise by introducing a *skeletal graph* (Sec. IV-B)), based on which we define a group of primitive evolution operations for nodes and clusters, and introduce their algebra for incremental tracking (Sec. V);
- We leverage the incremental computation by proposing two algorithms based on bulk updating: ICM for the incremental cluster maintenance and eTrack for the cluster evolution tracking, respectively (Sec. VI);
- Our application on event evolution tracking in large Twitter streams demonstrates that our framework can effectively track all kinds of cluster evolution patterns from highly dynamic networks in real time (Sec. VII).

More related work is discussed in Sec. VIII. The problem is formalized in Sec. II. For convenience, we summarize the major notations used in this paper in Table I.

II. PROBLEM FORMALIZATION

We formally define dynamic network and dynamic clusters here, and then introduce the problem this paper seeks to solve.

Dynamic Network. A dynamic network is a network with node and edge updates over time. We define a snapshot of an dynamic network at moment t as a weighted graph $G_t(V_t, E_t)$, where an edge $e(u, v) \in E_t$ connects nodes u, v in V_t and $s(u, v)$ is the similarity between them. For the problem studied in this paper, we assume a dynamic network is the input and $s(u, v) \in (0, 1]$ is a value usually set by a specific similarity function. From time t to $t+1$, we use ΔG_{t+1} to describe the updating subgraph applied to G_t, i.e., $G_t + \Delta G_{t+1} = G_{t+1}$. Naturally, a dynamic network $G_{1:i}$ from moment 1 to i can be described as a continuous updating subgraph sequence applied at each moment. The formal definition is given below.

***Definition** 1:* A *dynamic network* $G_{i:j}$ from moment i to j is denoted as $(G_i; \Delta G_{i+1}, \cdots, \Delta G_j)$, where $G_i(V_i, E_i)$ is a weighted graph with node set V_i and edge set E_i, and ΔG_{t+1}

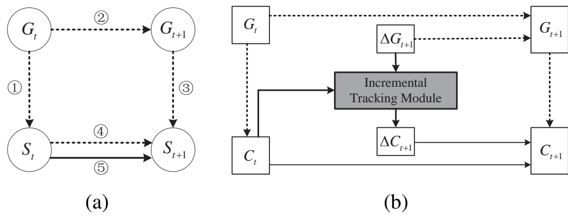


Fig. 2. (a) The commutative diagram between dynamic networks G_t, G_{t+1} and cluster sets S_t, S_{t+1}. The "divide-and-conquer" baseline and our *Incremental Tracking* are annotated by solid and dotted lines respectively. (b) The workflow of incremental tracking module, which shows our framework tracks cluster evolution dynamics by only consuming the updating subgraph ΔG_{t+1}.

$(i \leq t < j)$ is an updating subgraph at moment $t+1$ such that $G_t + \Delta G_{t+1} = G_{t+1}$.

When the network evolves from G_t to G_{t+1}, we reasonably assume that at moment $t+1$, only a small portion of G_t is incrementally updated, i.e., $|V_{t+1} - V_t| + |V_t - V_{t+1}| \ll |V_t|$ and $|E_{t+1} - E_t| + |E_t - E_{t+1}| \ll |E_t|$. This assumption generally holds in practice, and when it doesn't, we can shorten moment interval sufficiently to make the assumption hold. For simplicity, we express ΔG_{t+1} as a sequence of node additions and deletions, e.g., $\Delta G_{t+1} := +v_1 - v_2$ means adding node v_1 and all the edges incident with v_1 in a single operation, and analogously, deleting node v_2 and its incident edges in a subsequent operation.

Dynamic Clusters. Let's suppose that C_t is a subgraph in G_t, and $isCluster(C_t)$ is a boolean function to validate whether C_t is a cluster or not, with the exact definition given in Sec. IV-C. In the following, we define a dynamic density cluster.

***Definition** 2:* A *dynamic cluster* $C_{i:j}$ from moment i to j is denoted as $(C_i; \Delta C_{i+1}, \cdots, \Delta C_j)$ where $isCluster(C_i) = True$, and ΔC_{t+1} $(i \leq t < j)$ is an updating subgraph at moment $t+1$ that makes $C_t + \Delta C_{t+1} = C_{t+1}$ and $isCluster(C_{t+1}) = True$.

The Problem. We focus on addressing the following problem:

***Problem** 1:* Supposing $G_{i:j} = (G_i; \Delta G_{i+1}, \cdots, \Delta G_j)$ is a large dynamic network and $isCluster(C_t)$ is a binary validation function for cluster candidate C_t, the problem of incremental cluster evolution is to generate an updating subgraph sequence $(\Delta C_{i+1}, \cdots, \Delta C_j)$ with $C_t + \Delta C_{t+1} = C_{t+1}$ and $isCluster(C_{t+1}) = True$, where $i \leq t < j$.

The cluster evolution patterns can be observed from the updating sequence. For example, if $C_t \neq \emptyset$ but $C_{t+1} = \emptyset$, it means C_t dies at moment $t+1$. $C_t = \emptyset$ but $C_{t+1} \neq \emptyset$, a new cluster C_{t+1} is born at moment $t+1$. Typical cluster evolution patterns include birth, death, growth, decay, merge and split. In this paper, we aim to track the complete set of cluster evolution patterns in real time.

III. Incremental Tracking Framework 0.5

We illustrate the relationship between dynamic networks G_t, G_{t+1} and cluster sets S_t, S_{t+1} at consecutive moments as a commutative diagram in Figure 2(a). The traditional approaches for tracking dynamic network related problems usually follow a "divide-and-conquer" spirit [7], [8], which consists of three components: (1) decompose a dynamic network into a series of snapshots for each moment, (2) apply graph mining algorithms on each snapshot to find useful patterns, (3) match patterns between different moments to generate a dynamic pattern sequence. Applied to our problem, to track cluster evolution patterns, these steps are:

Step ①: At moment t, identify the cluster set S_t from G_t;

Step ②: At moment $t+1$, as the network evolves, generate G_{t+1} from G_t using ΔG_{t+1};

Step ③: Again, identify the cluster set S_{t+1} from G_{t+1};

Step ④: Generate cluster evolution patterns from time t to $t+1$ by tracing the correspondence between S_t and S_{t+1}.

However, this approach suffers from both performance and quality. Firstly, repeated extraction of clusters from large networks from scratch is a very expensive operation (steps ① and ③), and tracing the correspondence between cluster sets at successive moments is also expensive (step ④). Secondly, the step of tracing correspondence, since it is done after two cluster sets are generated, may lead to loss of accuracy. In contrast, the method we propose is incremental tracking of cluster evolution, which corresponds to step ⑤ in Figure 2(a). The workflow of this incremental tracking from moment t to $t+1$ is illustrated in Figure 2(b). More precisely, for the very first snapshot of the dynamic network, say G_0, our approach will generate the corresponding event set S_0 from scratch. After this, this step is never applied again. In the steady state, we only apply step ⑤, i.e., we incrementally derive S_{t+1} from S_t and ΔG_{t+1}. The experiments on real data set show that our incremental tracking approach outperforms the traditional baselines in both performance and quality.

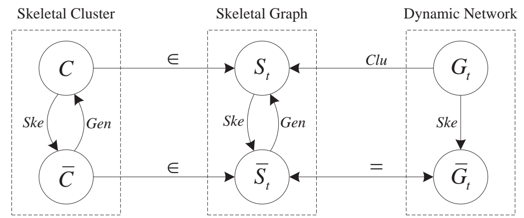


Fig. 3. The functional relationships between different types of objects defined in this paper, e.g., the arrow from G_t to $\overline{G}_t$ with label Ske means $\overline{G}_t = Ske(G_t)$. Refer to Table I for notations.

IV. Skeletal Graph Clustering

The functional relationships between different types of objects defined in this paper are illustrated in Figure 3. As an example, the arrow from G_t to $\overline{G}_t$ with label Ske means $\overline{G}_t$ is derived from G_t by function Ske, i.e., $\overline{G}_t = Ske(G_t)$. See Table I for notations used. The various objects and their relationships will be explained in the rest of the paper.

A. Post Network Construction

Our approach is based on constructing a network of posts and maintaining the network over a moving time window, as posts stream in and fade out. This network is used for subsequent analysis. In this section, we describe the construction.

Social Stream Preprocessing. Social posts such as tweets are usually written in an informal way. Simply treating each post text as a bag of words [14] will lead to loss of accuracy, since different words have different weights in deciding the post's topic. To design a processing strategy that can quickly and robustly extract the topic of a post, we focus on the entity words. However, traditional Named Entity Recognition tools [15] only support a narrow range of entities like Locations, Persons and Organizations. NLP parser based approaches [16] are not appropriate due to the informal writing style of posts and the need for high processing speed. To broaden the applicability, we treat each noun in the post text as a candidate entity. Technically, we obtain nouns from a post text using a Part-Of-Speech Tagger[1], and if a noun is plural (POS tag "NNS" or "NNPS"), we obtain its singular form. In practice, we find this preprocessing technique to be robust and efficient. In the Twitter dataset we used in experiments (see Section VII), each tweet contains 4.9 entities on an average. We formalize a post p as a triple (L, τ, u), where p^L is the list of entities, p^τ is the time stamp, and p^u is the author.

Fading Similarity. Traditional similarity measures such as TF-IDF based cosine similarity, Jaccard Coefficient and Pearson Correlation [14] only consider the post content. However, clearly time stamps should play an important role in determining post similarity, since posts created closer together in time are more likely to discuss the same event. We introduce the notion of fading similarity to capture both content similarity and time proximity. E.g., with Jaccard coefficient as the underlying content similarity measure, the *fading similarity* is defied as

$$S_F(p_i, p_j) = \frac{|p_i^L \cap p_j^L|}{|p_i^L \cup p_j^L| \cdot e^{|p_i^\tau - p_j^\tau|}} \tag{1}$$

. where we use an exponential function to incorporate the decaying effect of time lapse between the posts. It is trivial to see that $0 \leq S_F(p_i, p_j) \leq 1$ and that $S_F(p_i, p_j)$ is symmetric.

Post Network. To find the correlation between posts, we build a post network $G_t(V_t, E_t)$ based on the following rule: if the fading similarity between two posts (p_i, p_j) is higher than a given threshold λ, we create an edge $e(p_i, p_j)$ between them and set the edge similarity $s(p_i, p_j) = S_F(p_i, p_j)$. Obviously, a lower λ retains more semantic similarities but results in much higher computation cost, and we set $\lambda = 0.3$ empirically on Twitter streams to gain a balance between edge sparsity and information richness. Consider a time window of observation and consider the post network at the beginning. While we move forward in time and new posts appear and old posts fade out, $G_t(V_t, E_t)$ is dynamically updated at each moment, with new nodes/edges added and old nodes/edges removed. On the scale of Twitter streams with millions of tweets per hour, $G_t(V_t, E_t)$ is truly a large and fast dynamic network.

Linkage Search. Removing a node and associated edges from $G_t(V_t, E_t)$ is an easy operation. In contrast, when a new post p_i appear, it is impractical to compare p_i with each node p_j in V_t to verify the satisfaction of $S_F(p_i, p_j) > \lambda$, since the node size $|V_t|$ can easily go up to millions. To solve this problem, first we construct a post-entity bipartite graph, and then perform a two-step random walk process to get the hitting counts. The main idea of linkage search is to let a random surfer start from post node p_i and walk to any entity node in p_i^L on the first step, and continue to walk back to posts except p_i on the second step. All the posts visited on the second step form the candidates of p_i's neighbors. Supposing the average number of entities in each post is d_1 and the average number of posts mentioning each entity is d_2, then linkage search can find the neighbor set of a given post in time $O(d_1 d_2)$. In our Twitter dataset, d_1 and d_2 are usually below 10, which supports the construction of a post network on the fly.

[1]POS Tagger, http://nlp.stanford.edu/software/tagger.shtml

B. Node Prioritization

In reality, many posts tend to be just noise, so it is essential to identify those nodes that play a central role in describing clusters. On web link graph analysis, there is a lot of research on node authority ranking, e.g., HITS and PageRank [14]. However, most of these methods are iterative and not applicable to the *single-pass* computation on streaming data. Node prioritization is a technique to quickly differentiate and rank the processing order of nodes by their roles in a single pass. It is extremely useful in big graph mining, where there are too many nodes to be processed and many of them are of little significance. However, to the best of our knowledge, there is a lack of the study on single-pass node prioritization in a streaming environment.

In this paper, we perform node prioritization based on density parameters (ε, δ), where $0 < \varepsilon < 1$, and $\varepsilon \leq \delta$. In density-based clustering (e.g., DBSCAN [17]), the threshold $MinPts$ is used as the minimum number of nodes in an ε-neighborhood, required to form a cluster. We adapt this and use a weight threshold δ as the minimum total weight of neighboring nodes, required to form a cluster. The reason we choose density-based approaches is that, compared with partitioning-based approaches (e.g., K-Means [18]) and hierarchical approaches (e.g., BIRCH [18]), density-based methods such as DBSCAN define clusters as areas of higher density than the remainder of the data set, which is effective in finding arbitrarily-shaped clusters and is robust to noise. Moreover, density-based approaches are easy to adapt to support single-pass clustering. In the post network, we consider ε to be a *similarity threshold* to decide connectivity, and can be used to define the post priority.

Definition 3: Given a post $p = (L, \tau, a)$ in post network $G_t(V_t, E_t)$ and similarity threshold ε, the *priority* of p at moment t $(t \geq p^\tau)$, is defined as

$$w^t(p) = \frac{1}{e^{|t - p^\tau|}} \sum_{q \in \mathcal{N}(p)} S_F(p, q) \tag{2}$$

where $\mathcal{N}(p)$ is the subset of p's neighbors with $S_F(p, q) > \varepsilon$.

Notice that post priority decays as time moves forward. Thus, post priority needs to be continuously updated. In practice, we only store the sum $\sum_{q \in \mathcal{N}(p)} S_F(p, q)$ with p to avoid frequent updates and compute $w^t(p)$ on demand.

Skeletal Graph. With post priority computed, we use δ as a *priority threshold* to differentiate nodes in $G_t(V_t, E_t)$:

- A post p is a *core post* if $w^t(p) \geq \delta$;
- It is a *border post* if $w^t(p) < \delta$ but there exists at least one core post $q \in \mathcal{N}(p)$;
- It is a *noise post* if it is neither core nor border, i.e., $w^t(p) < \delta$ and there is no core post in $\mathcal{N}(p)$.

Intuitively, a post is a core post if it shares enough common entities with many other posts. Neighbors of a core post are at least border posts, if not core posts themselves. Core posts play a central role: if a core post p is found to be a part of a cluster C, its neighboring (border or core) posts will also be a part of C. This property can be used in the single-pass clustering: if an incoming post p is "reachable" from an existing core post q, post p will be assigned to the cluster with q. Core posts connected by edges with similarity higher than ε will form a summary of $G_t(V_t, E_t)$, that we call the skeletal graph.

***Definition** 4:* Given post network $G_t(V_t, E_t)$ and density parameters (ε, δ), we define the *skeletal graph* as the subgraph of $G_t(V_t, E_t)$ induced by posts with $w^t(p) \geq \delta$ and edges with similarity higher than ε. We write $\overline{G}_t = Ske(G_t)$.

Ideally, $\overline{G}_t(\overline{V}_t, \overline{E}_t)$ will retain important information in G_t. Empirically, we found that adjusting the granularity of (ε, δ) to make the size $|\overline{V}_t|$ roughly equal to 20% of $|V_t|$ leads to a good balance between the quality of the skeletal graph in terms of the information retained and its space complexity. More tuning details can be found in Section VII-A.

C. Skeletal Cluster Identification

One of the key ideas in our incremental cluster evolution tracking approach is to use the updating subgraph ΔG_{t+1} between successive moments to maintain the skeletal clusters. Post clusters are constructed from these skeletal clusers. Maintaining skeletal clusters can be done efficiently since the skeletal graph is much smaller in size than the post graph it's obtained from. Besides efficiency, skeletal cluster has the advantage of giving the correspondence between successive post clusters in a very small cost.

***Definition** 5:* Given $G_t(V_t, E_t)$ and the corresponding skeletal graph $\overline{G}_t(\overline{V}_t, \overline{E}_t)$, a *skeletal cluster* $\overline{C}$ is a connected component of $\overline{G}_t$. A *post cluster* is a set of core posts and border posts generated from a skeletal cluster $\overline{C}$, written as $C = Gen(\overline{C})$, using the following expansion rules:

- All posts in $\overline{C}$ form the core posts of C.
- For every core post in C, all its neighboring border posts in G_t form the border posts in C.

In what follows, by cluster, we mean a post cluster, distinguished from the explicit term skeletal cluster. By definition, a core post only appears in one (post) cluster. If a border post is associated with multiple core posts in different clusters, this border post will appear in multiple (post) clusters.

V. Incremental Cluster Evolution

In this section, we discuss the incremental evolution of skeletal graph and post clusters under the fading time window, which forms the theoretical basis for the algorithms in Sec. VI.

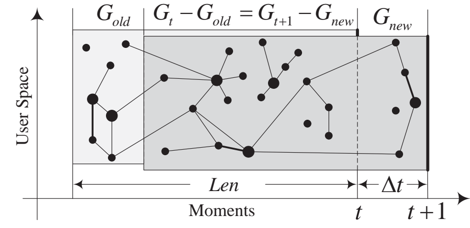


Fig. 4. An illustration of the fading time window from time t to $t+1$, where post priority may fade w.r.t. the end of time window. G_t will be updated by deleting subgraph G_{old} and adding subgraph G_{new}.

A. Fading Time Window

Fading (or decay) function and sliding time window are two common aggregation schemes used in time-evolving graphs (e.g., see [9]). Fading scheme puts a higher emphasis on newer posts, as captured by fading similarity in Eq. (1). Sliding time window scheme (posts are first-in, first-out) is essential because it provides a scope within which a user can monitor and track the evolution. Since clusters evolve quickly from moment to moment, even within a given time window, it is important to highlight new posts and degrade old posts using the fading scheme. Thus, we combine these two schemes and introduce a *fading time window*, as illustrated in Figure 4. In practice, users can specify the length of the time window to adjust the scope of monitoring. Users can also choose different fading functions to penalize old posts and highlight new posts in different ways. Let Δt denote the time interval. For simplicity, we abbreviate the moment $(t+i \cdot \Delta t)$ as $(t+i)$. When the time window slides from moment t to $t+1$, the post network $G_t(V_t, E_t)$ will be updated to be $G_{t+1}(V_{t+1}, E_{t+1})$. Suppose $G_{old}(V_{old}, E_{old})$ is the old subgraph (of G_t) that lapses at moment $t+1$ and $G_{new}(V_{new}, E_{new})$ is the new subgraph (of G_{t+1}) that appears (see Figure 4). Clearly,

$$G_{t+1} = G_t - G_{old} + G_{new} \tag{3}$$

Let Len be the time window length. We assume $Len > 2\Delta t$, which makes $V_{old} \cap V_{new} = \emptyset$. This assumption is reasonable in applications, e.g., we set Len to 1 week and Δt to 1 day.

B. Network Evolution Operations

We analyze the evolution process of networks and clusters at each moment and abstract them into five primitive operators: $+$, $-$, $\odot$, $\uparrow$, $\downarrow$. We classify the operators based on the objects they manipulate: nodes or clusters, and define them below.

***Definition** 6:* Primitive node operations:

- $G_t + p$: add a new post p into $G_t(V_t, E_t)$ where $p \notin V_t$. All the new edges associated with p will be constructed automatically by linkage search (explained in Sec. IV-A);
- $G_t - p$: delete a post p from $G_t(V_t, E_t)$ where $p \in V_t$. All the existing edges associated with p will be automatically removed from E_t.
- $\odot G_t$: update the post priority scores in G_t.

Composite node operations:

- $G_t \oplus p = \odot(G_t + p)$: add a post p into $G_t(V_t, E_t)$ where $p \notin V_t$ and update the priority of related posts;

- $G_t \ominus p = \odot(G_t - p)$: delete a post p from $G_t(V_t, E_t)$ where $p \in V_t$ and update the priority of related posts.

Definition *7:* Primitive cluster evolution operations:

- $+C$: generate a new cluster C;
- $-C$: remove an old cluster C;
- $\uparrow(C, p)$: increase the size of C by adding post p;
- $\downarrow(C, p)$: decrease the size of C by removing post p.

Composite cluster evolution operations:

- $Merge(S) = +C - S$: merge a set of clusters S into a new single cluster C and remove S;
- $Split(C) = -C + S$: split a single cluster C into a set of new clusters S and remove C.

In particular, composite node operations are designed to conveniently describe the adding/deleting of posts with priority scores updated in the same time, and composite cluster operations are designed to capture the advanced evolution patterns of clusters. Each operator defined above on a single object can be extended to a set of objects, i.e., for a node set $\mathcal{X} = \{p_1, p_2, \cdots, p\}$, $G_t + \mathcal{X} = G_t + p_1 + p_2 + \cdots + p$. This is well defined since $+$ is associative and commutative. We use the left-associative convention for '$-$': that is, we write $A - B - C$ to mean $(A - B) - C$. These operators will be used later in the formal description of the evolution procedures. Figure 5(a) depicts the role played by the primitive operators in the tracking of cluster evolutions from dynamic networks.

C. Skeletal Graph Evolution Algebra

The updating of skeletal graphs from $\overline{G}_t$ to $\overline{G}_{t+1}$ is the core task in cluster evolution tracking. If we ignore the node priorities for a moment, the following formula shows different ways to compute the overlapping part in G_{t+1} and G_t, as illustrated in Figure 4(b):

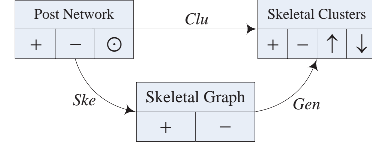
$$G_{t+1} - G_{new} = G_t - G_{old} = G_{t+1} \ominus G_{new} = G_t \ominus G_{old} \quad (4)$$

However, at the skeletal graph level, $Ske(G_{t+1} - G_{new}) \neq Ske(G_t - G_{old})$: some core posts in $G_t - G_{old}$ may no longer be core posts due to the removal of edges incident with nodes in G_{old} or simply due to the passing of time; some non-core posts may become core posts because of the adding of edges with nodes in G_{new}. To measure the changes in the overlapping part, we define the following three components.

Definition *8:* Updated components in overlap:

- $\overline{S}_+ = Ske(G_{t+1} - G_{new}) - Ske(G_{t+1} \ominus G_{new})$: components of non-core posts in $G_t - G_{old}$ that become core posts in $G_{t+1} - G_{new}$ due to the adding of G_{new};
- $\overline{S}_- = Ske(G_t - G_{old}) - Ske(G_t \ominus G_{old})$: components of core posts in $G_t - G_{old}$ that become non-core posts in $G_{t+1} - G_{new}$ due to the removing of G_{old};
- $\overline{S}_\odot = Ske(G_t \ominus G_{old}) - Ske(G_{t+1} \ominus G_{new})$: components of core posts in $G_t - G_{old}$ that become non-core posts in $G_{t+1} - G_{new}$ due to the passing of time.

Based on Definition 8, from moment t to $t+1$, the changes of core posts in the overlapping part, i.e., $G_{t+1} - G_{new}$



(a)

$\lvert N_c(p) \rvert$	0	1	≥ 2
Add a core post p	$+$	$\uparrow$	$Merge$
Delete a core post p	$-$	$\downarrow$	$Split$

(b)

Fig. 5. (a) The relationships between primitives and evolutions. Each box represents an evolution object and the arrows between them describe inputs/outputs. (b) The evolutionary behavior table for clusters when adding or deleting a core post p.

(equivalently, $G_t - G_{old}$ – see Figure 4), can be updated using the components $\overline{S_+}$, $\overline{S_-}$ and $\overline{S_\odot}$. That is,

$$\begin{aligned} & Ske(G_{t+1} - G_{new}) - Ske(G_t - G_{old}) \\ = \; & (Ske(G_{t+1} - G_{new}) - Ske(G_{t+1} \ominus G_{new})) \\ & -(Ske(G_t - G_{old}) - Ske(G_t \ominus G_{old})) \\ & -(Ske(G_t \ominus G_{old}) - Ske(G_{t+1} \ominus G_{new})) \\ = \; & \overline{S_+} - \overline{S_-} - \overline{S_\odot} \end{aligned} \quad (5)$$

Let $\overline{S}_{old}$ and $\overline{S}_{new}$ denote the sets of skeletal clusters in G_{old} and G_{new} respectively. The following theorem characterizes the iterative and incremental updating of skeletal graphs from moment t to $t+1$, and it plays a central role in the cluster evolution.

Theorem *1:* From moment t to $t+1$, the skeletal graph evolves by removing core posts in G_{old}, adding core posts in G_{new} and updating core posts in the overlapping part. That is

$$\overline{S}_{t+1} = \overline{S}_t - \overline{S}_{old} - \overline{S}_- - \overline{S}_\odot + \overline{S}_{new} + \overline{S}_+ \quad (6)$$

Proof Sketch: Since operator '$-$' does not update post priority, we have $Ske(G_{t+1} - G_{new}) = Ske(G_{t+1}) - Ske(G_{new}) = \overline{S}_{t+1} - \overline{S}_n$, $Ske(G_t - G_{old}) = Ske(G_t) - Ske(G_{old}) = \overline{S}_t - \overline{S}_{old}$. Then, $\overline{S}_{t+1} - \overline{S}_{new} - \overline{S}_t + \overline{S}_{old} = \overline{S}_+ - \overline{S}_- - \overline{S}_\odot$ and we get the conclusion. □

Theorem 1 indicates that we can incrementally maintain skeletal clusters $\overline{S}_{t+1}$ from $\overline{S}_t$. Since we define (post) clusters based on skeletal clusters, this incremental updating of skeletal clusters benefits incremental updating of cluster evolution essentially.

D. Incremental Cluster Evolution

Let $S_t = Clu(G_t)$ denote the set of clusters obtained from the post network G_t. Notice that noise posts in G_t do not appear in any clusters, so the number of posts in S_t is typically smaller than $|V_t|$. Next, we explore the incremental cluster evolution problem from two levels: the node-by-node updating level and subgraph-by-subgraph updating level.

Node-by-Node Evolution. The basic operations underlying cluster evolution are the cases when S_t is modified by the addition or deletion of a cluster that includes only one post. In the following, we analyze and show the evolution of clusters by adding or deleting a post p. When adding p, we let $N_c(p)$

denote the set of clusters that p's neighboring core posts belong to *before* p is added. When deleting p, let $N_c(p)$ denote the set of clusters that p's neighboring core posts belong to *after* p is removed. $|N_c(p)| = 0$ means p has no neighboring core posts. Notice that $Merge$ and $Split$ are composite operations and can be decomposed into a series of cluster primitive operations. We show the evolution behaviors of clusters in Figure 5(b) and explain the detail below.

(a) Addition: $S_t + \{p\}$

If p is a noise post after being added into G_t, ignore p. If p is a border post, add p to each cluster in $N_c(p)$. Else, p is a core post and we do the following:

- If $|N_c(p)| = 0$: apply $+C$, where $C = \{p\} \cup \mathcal{N}(p)$;
- If $|N_c(p)| = 1$: apply $\uparrow (C, \{p\} \cup \mathcal{N}(p))$, where C is the lone cluster in $N_c(p)$;
- If $|N_c(p)| \geq 2$: apply $Merge = +C - \sum_{C' \in N_c(p)} C'$ and $C = N_c(p) \cup \{p\} \cup \mathcal{N}(p)$.

(b) Deletion: $S_t - \{p\}$

If p is a noise post before being deleted from G_t, ignore p. If p is a border post, delete p from each cluster in $N_c(p)$. Else, p is a core post and we do the following:

- If $|N_c(p)| = 0$: apply $-C$ where $p \in C$;
- If $|N_c(p)| = 1$: apply $\downarrow (C, \{p\} \cup \mathcal{N}(p))$;
- If $|N_c(p)| \geq 2$: apply $Split = -C + \sum_{C' \in N_c(p)} C'$, where $p \in C$ before the deletion.

Subgraph-by-Subgraph Evolution. When dynamic networks such as post networks in social streams surge quickly, the node-by-node processing for cluster evolution will lead to a poor performance. To accelerate the performance, we consider the subgraph-by-subgraph updating approach. Let $Clu(G_{new}) = S_{new}$ and $Clu(G_{old}) = S_{old}$ be the cluster sets of the graphs G_{new} and G_{old}, and S_t be the set of all clusters at moment t. As the time window moves forward to moment $t+1$, if we add G_{new} to the network G_t, clusters will evolve as follows:

$$\begin{aligned} & Clu(G_t + G_{new}) = Gen(Ske(G_t + G_{new})) \\ = \; & Gen(Ske(G_t) + Ske(G_{new}) + \overline{S}_+ - \overline{S}_\odot) \quad \text{(Definition 8)} \\ = \; & S_t + S_{new} + S_+ - S_\odot \qquad (7) \end{aligned}$$

where $S_+ = Gen(\overline{S}_+)$ and $S_\odot = Gen(\overline{S}_\odot)$. Similarly, if we remove G_{old} from the network G_t, clusters evolve as follows:

$$\begin{aligned} & Clu(G_t - G_{old}) = Gen(Ske(G_t - G_{old})) \\ = \; & Gen(Ske(G_t) - Ske(G_{old}) - \overline{S}_-) \quad \text{(Definition 8)} \\ = \; & S_t - S_{old} - S_- \qquad (8) \end{aligned}$$

where $S_- = Gen(\overline{S}_-)$. Based on Equation (7) and (8), from moment t to $t+1$, the set of clusters can be incrementally updated by the iterative computation

$$\begin{aligned} S_{t+1} &= Clu(G_{t+1}) = Clu(G_t - G_{old} + G_{new}) \\ &= S_t - S_{old} - S_- + S_{new} + S_+ - S_\odot \qquad (9) \end{aligned}$$

Equation (9) can be also verified by applying Gen function on both sides of Equation (6). Naturally, Equation (9) provides a theoretical basis for the incremental computation of cluster evolution: as the post network evolves from G_t to G_{t+1}, we do not compute S_{t+1} from G_{t+1}. Instead, we incrementally update S_t by means of the five cluster sets appearing in Equation (9), using simple set operations. Since the sizes of G_{old} and G_{new} are usually very small compared with G_t, these five cluster sets are also of small size and so we can generate S_{t+1} quickly from them. The details of incremental computation are discussed in Section VI.

VI. Incremental Algorithms

The traditional approach of decomposing an evolving graph into a series of snapshots suffers from both quality and performance, since clusters are generated from scratch and matched heuristically at each moment. To overcome this limitation, we propose an incremental tracking framework, as introduced in Section V and illustrated in Figure 2(b). In this section, we leverage our incremental computation by proposing Algorithm 1 for the incremental cluster maintenance (ICM) and Algorithm 2 for the cluster evolution tracking (eTrack) respectively. Since at each moment $|V_{old}| + |V_{new}| \ll |V_t|$, our algorithms can save a lot computation by adjusting clusters incrementally, rather than generating them from scratch.

Bulk Updating. Traditional incremental computation on dynamic graphs usually treats the addition/deletion of nodes or edges one by one [19], [20]. However, in a real scenario, since social posts arrive at a high speed, the post-by-post incremental updating will lead to very poor performance. In this paper, we speed up the incremental computation of S_t by *bulk updating*. Clearly, updating S_t with a single node $\{p\}$ is a special case of bulk updating. Here, a bulk corresponds to a cluster of posts and we "lift" the post-by-post updating of S_t to the bulk updating level. Recall that $N_c(p)$ is the neighboring cluster set of p where p is a core post. To understand the bulk updating in Algorithm 1, for a cluster C, we define $N_c(\overline{C})$ as the neighboring cluster set of posts in $\overline{C}$, i.e., $N_c(\overline{C}) = \cup_{p \in \overline{C}} N_c(p)$ where $\overline{C} = Ske(C)$. When C is added into or deleted from S_t as a bulk, the size of $N_c(\overline{C})$ will decide the evolution patterns of clusters from moment t to $t+1$ after C is added or deleted, as shown in Figure 5(b). Since C is usually a small subgraph, we consider a bulk operation can be done in constant time.

Incremental Cluster Maintenance (ICM). The steps for incremental cluster maintenance (ICM) from any moment t to $t+1$ are summarized in Algorithm 1. The ICM algorithm follows the iterative computation shown in Equation (9), that is $S_{t+1} = S_t - S_{old} - S_- - S_\odot + S_{new} + S_+$. As analyzed in Section V-D, each bulk addition and bulk deletion has three possible evolution behaviors, decided by the size of $N_c(\overline{C})$. Lines 3-13 deal with deleting a bulk C, where three patterns $\{-, \downarrow, Split\}$ are handled. Lines 15-26 deal with adding a bulk C and handle another three patterns $\{+, \uparrow, Merge\}$. Supposing there are n bulk updates in ICM, the time complexity of ICM is $O(n)$. Since a bulk operation is generally completed in constant time, ICM is an efficient single-pass incremental computation algorithm.

Algorithm 1: ICM: Incremental Cluster Maintenance

Input: S_t, S_{old}, S_{new}, S_-, S_+, $S_\odot$
Output: S_{t+1}

1 $S_{t+1} = S_t$;
   // Delete $S_{old} \cup S_-$
2 **for** *each cluster* C *in* $S_{old} \cup S_- \cup S_\odot$ **do**
3   $\overline{C} = Ske(C)$;
4   $N_c(\overline{C}) = \cup_{p \in \overline{C}} N_c(p)$;
5   **if** $|N_c(\overline{C})| = 0$ **then**
6   remove cluster C from S_{t+1};
7   **else if** $|N_c(\overline{C})| = 1$ **then**
8   delete C from cluster C' where $C' \in N_c(\overline{C})$;
9   **else**
10   remove the cluster that C belongs to from S_{t+1};
11   **for** *each cluster* $C' \in N_c(\overline{C})$ **do**
12   assign a new cluster id for C';
13   add C' into S_{t+1};
   // Add $S_{new} \cup S_+$
14 **for** *each cluster* C *in* $S_{new} \cup S_+$ **do**
15   $\overline{C} = Ske(C)$;
16   $N_c(\overline{C}) = \cup_{p \in \overline{C}} N_c(p)$;
17   **if** $|N_c(\overline{C})| = 0$ **then**
18   assign a new cluster id for C and add C to S_{t+1};
19   **else if** $|N_c(\overline{C})| = 1$ **then**
20   add C into cluster C' where $C' \in N_c(\overline{C})$;
21   **else**
22   assign a new cluster id for C;
23   **for** *each cluster* $C' \in N_c(\overline{C})$ **do**
24       $C = C \cup C'$;
25   remove C' from S_{t+1};
26   add C into S_{t+1};
27 return S_{t+1};

Cluster Evolution Tracking (eTrack). Given a dynamic network $G_{i:j}$ and the set of clusters S_i at the start moment i, the eTrack algorithm will track the primitive cluster evolution operations at each moment, working on top of the ICM algorithm (Line 3). We summarize the steps of eTrack in Alg. 2. Basically, eTrack monitors the changes of clusters effected by ICM at each moment. If the cluster is not changed, eTrack will take no action; otherwise, eTrack will determine the corresponding cases and output the cluster evolution patterns (Lines 4-12). Notice that in Lines 5-8, if a cluster C in S_t has ClusterId id, we use the convention that $S_t(id) = C$ to access C by id, and $S_t(id) = \emptyset$ means there is no cluster in S_t with ClusterId id. Especially, lines 7-8 mean a cluster in S_t evolves into a cluster in S_{t+1} by deleting the posts in $S_t(id) - S_{t+1}(id)$ first and adding the posts in $S_{t+1}(id) - S_t(id)$ later. As an efficient monitoring algorithm, once we get S_{t+1} incrementally by ICM, the time complexity of eTrack is linear in the number of clusters in S_t and S_{t+1} at each moment.

VII. Experiments

In this section, we first discuss how to tune the construction of post network and skeletal graph to find the best selection

Algorithm 2: eTrack: Cluster Evolution Tracking

Input: $G = \{G_i, G_{i+1}, \cdots, G_j\}$, S_i
Output: Primitive cluster evolution operations

1 **for** t *from* i *to* j **do**
2   obtain S_{old}, S_{new}, S_-, S_+ from $G_{i+1} - G_i$;
3   $S_{t+1} = ICM(S_t, S_{old}, S_{new}, S_-, S_+, S_\odot)$;
4   **for** *each cluster* $C \in S_{t+1}$ **do**
5   id = $ClusterId(C)$;
6   **if** $S_t(id) \neq \emptyset$ **then**
7   output $\downarrow (C, S_t(id) - S_{t+1}(id))$;
8   output $\uparrow (C, S_{t+1}(id) - S_i(id))$;
9   **else** $+C$;
10   **for** *each cluster* $C \in S_i$ **do**
11   id = $ClusterId(C)$;
12   **if** $S_{t+1}(id) = \emptyset$ **then** $-C$;

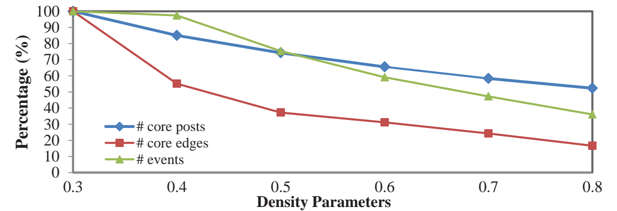


Fig. 6. The trends of the number of core posts, core edges and events when increasing δ from 0.3 to 0.8. We set $\delta = \varepsilon = 0.3$ as the 100% basis.

of entity extraction and density parameters. Then, we test the quality and performance of cluster evolution tracking algorithms on two social streams: Tech-Lite and Tech-Full that we crawled from Twitter. Our event detection baseline covers the major techniques reported in [1], [2], [3], [4]. Our evolution tracking baseline captures the essence of the state of the art algorithms reported in [7], [8]. All experiments are conducted on a computer with Intel 2.66 GHz CPU, 4 GB RAM. All algorithms are implemented in Java. We use the graph database Neo4J[2] to store and manipulate the post network.

Datasets. All datasets are crawled from Twitter via Twitter API. Although our cluster evolution tracking algorithm works regardless of the domain, in order to facilitate evaluation, we make the dataset domain specific. The crawling of datasets is performed as follows. We built a technology domain dataset called Tech-Lite by aggregating all the timelines of users listed in the Technology category of "Who to follow"[3] and their retweeted users. Tech-Lite has 352,328 tweets, 1402 users and the streaming rate is about 11700 tweets/day. Based on the intuition that the followees of users in Technology category are most likely to be in the same domain, we obtained a larger technology social stream called Tech-Full by collecting all the timelines followed by users in the Technology category. Tech-Full has 5,196,086 tweets, created by 224,242 users, whose streaming rate is about 7216 tweets/hour. Both Tech-Lite and Tech-Full include retweets and have a time span from Jan. 1 to Feb. 1, 2012. Since each tweet corresponds to a node in the

[2]http://neo4j.org/
[3]http://twitter.com/who_to_follow/interests

(a) Results of different entity extraction approaches.

Methods	#edges	#coreposts	#coreedges	#events
Hashtags	182905	6232	28964	196
Unigrams	142468	15070	46783	430
POS-Tagger	357132	21509	47808	470

(b) Precision and recall of top 50 events.

Methods	Precision (major events)	Recall (major events)	Precision (G-Trends)
HashtagPeaks	0.40	0.30	0.25
UnigramPeaks	0.45	0.40	0.20
Louvain	0.60	0.55	0.75
eTrack	0.80	0.80	0.95

TABLE II
TUNING POST NETWORK.

post network, both Tech-Lite and Tech-Full produce highly dynamic networks. Notice that the performance of our single-pass incremental approach is mainly affected by the streaming rate, rather than the dataset size.

A. Tuning Post Network and Skeletal Graph

Post Preprocessing. As described in Section IV, we extract entities from posts by POS tagger. One alternative approach to entity extraction is using hashtags. However, only 11% of the tweets in our dataset have hashtags, which results in lots of posts in the dataset having no similarity score between them. Another approach is simply tokenizing tweets into unigrams and treating unigrams as entities, and we call it the "Unigrams" approach, as discussed in [2]. Table 2(a) shows the comparison of the three entity extraction approaches in the first time window of the Tech-Full social stream. If we use "Unigrams", obviously the number of entities is larger than other two approaches, but the number of edges between posts tends to be smaller, because tweets written by different users usually share very few common words even when they talk about the same event. The "Hashtags" approach also produces a smaller number of edges, core posts and events, since it generates a much sparser post network. Overall, the "POS-Tagger" approach can discover more similarity relationships between posts and produce more core posts and events given the same social stream and parameter setting.

Density Parameters. The density parameters (ε, δ) control the construction of the skeletal graph. Clearly, the higher the density parameters, the smaller and sparser the skeletal graph. Figure 6 shows the number of core posts, core edges and events as a percentage of the numbers for $\varepsilon = 0.3$, as δ increases from 0.3 to 0.8. Results are obtained from the first time window of the Tech-Full social stream. We can see the rate of decrease of #events is higher than the rates of #core posts and #core edges after $\delta > 0.4$, because events are less likely to form in sparser skeletal graphs. More small events can be detected by lower density parameters, but the computational cost will increase because of larger and denser skeletal graphs. However, for big events, they are not very sensitive to these density parameters. We set $\varepsilon = 0.3$, $\delta = 0.5$ as a trade-off between the size and number of events one hand and processing efficiency on the other.

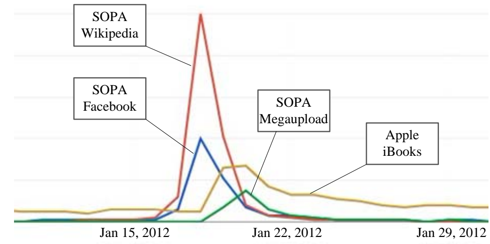


Fig. 7. Examples of Google Trends peaks in January 2012. We validate the events generated by cTrack by checking the existence of volume peaks at a nearby time moment in Google Trends. Although these peaks can detect bursty events, Google Trends cannot discover the merging/splitting patterns.

HashtagPeaks	UnigramPeaks	Louvain	CET
CES	google	Apple iphone ipad	CES conference
SOPA	ces	CES ultrabook tablet	SOPA PIPA
EngadgetCES	apple	Google search privacy	Hug new year
opengov	video	Week's Android games	RIM new CEO
gov20	sopa	Kindle Netflix app	Yahoo jerry yang
CES2012	twitter	Internet people time	Samsung Galaxy Nexus
PIPA	year	Hope weekend	Apple iBooks
opendata	facebook	SOPA Megaupload	Facebook IPO News
StartupAmerica	app	SOPA PIPA Wikipedia	Martin Luther King
win7tech	iphone	Facebook IPO	Tim Cook Apple stock

Fig. 8. Lists of top 10 events detected from Twitter Technology streams in January 2012 by baseline HashtagPeaks, UnigramPeaks, Louvain and our incremental tracking approach eTrack.

B. Cluster Evolution Tracking

Ground truth. To generate the ground truth, we crawl news articles in January 2012 from famous technology websites such as TechCrunch, Wired, CNET, etc, without looking at tweets. Then we treat the titles of news as posts and apply our event tracking algorithm to extract event evolution patterns. Finally, a total of 20 major events with life cycles are identified as ground truth. Typical events include "happy new year", "CES 2012", "sopa wikipedia blackout", etc. To find more small and less noticeable events, we use Google Trends for Search[4], which shows the traffic trends of keywords that appeared in Google Search along the time dimension. If an event-indicating phrase has a volume peak in Google Trends at a specific time, we say this event is sufficiently validated by the real world. We validate the correctness of an event C_i by the following process: we pick the top 3 entities of C_i ranked by frequency and search them in Google Trends, and if the traffic trend of these top entities has a distinct peak at a nearby time to C_i, we consider that C_i corresponds to a real world event widely witnessed by the public. Four examples of Google Trends peaks are shown in Figure 7. It is not surprising to find that the birth of events in social streams is usually earlier than its appearance in Google Trends.

Cluster Annotation. Considering the huge volume of posts in a cluster, it is important to summarize and present a post cluster as a conceptual event to aid human perception. In related work, Twitinfo [2] represents an event it discovers from Twitter by a timeline of tweets, showing the tweet activity by volume over time. However, it is tedious for users to read

[4]http://www.google.com/trends/

tweets one-by-one to figure out the event detail. In this paper, we summarize a snapshot of a cluster by a word cloud [21]. The font size of a word in the cloud indicates its popularity. Compared with Twitinfo, word cloud provides a summary of the cluster at a glance and is much easier for human to read.

Baseline 1: Peak-Detection. In recent works [1], [2], [3], [4], events are generally detected as volume peaks of phrases over time in social streams. These approaches share the same spirit that aggregates the frequency of event-indicating phrases at each moment to build a histogram and generates events by detecting volume peaks in the histogram. We design two variants of Peak-Detection to capture the major techniques used by these state-of-the-art approaches.

- Baseline 1a: HashtagPeaks which aggregates hashtags;
- Baseline 1b: UnigramPeaks which aggregates unigrams.

Notice, both baselines above are for event detection only. Lists of the top 10 events detected by HashtagPeaks and UnigramPeaks are presented in Figure 8. Some highly frequent hashtags like "#opengov" and "#opendata" are not designed for event indication, hurting the precision. UnigramPeaks uses the unigrams extracted from the social stream preprocessing stage, which has a better quality than HashtagPeaks. However, both of them are limited in their representation of events, because the internal structure of events is missing. Besides, although these peaks can detect bursty words, they cannot discover cluster evolution patterns such as the merging/splitting. For example, in Figure 7, there is no way to know "Apple announced iBooks" is a split from the big event "SOPA" earlier, as illustrated in detail in Figure 9.

Baseline 2: Community Detection. A community in a large network refers to a subgraph with dense internal connections and sparse connections with other communities. It is possible to define an event as a community of posts. Louvain method [22], based on modularity optimization, is the state-of-the-art approach community detection method in terms of performance. We design a baseline called "Louvain" to detect events defined based on post communities. The top 10 events generated by Louvain are shown in Figure 8. As we can see, not every result detected by the Louvain method is meaningful. For example, "Apple iphone ipad" and "Internet people time" are too vague to correspond to any concrete real events. The reason is, although Louvain method can make sure every community has relatively dense internal and sparse external connections, it cannot guarantee that every node in the community is important and has a sufficiently high connectivity with other nodes in the same community. It is highly possible that a low-degree node belongs to a community only because it has zero connectivity with other communities. Furthermore, noise posts are quite prevalent in Twitter and they negatively impact Louvain method.

Baseline 3: Pattern-Matching. We design a baseline to track the evolution patterns of clusters between snapshots. In graph mining, the "divide-and-conquer" approach of decomposing the evolving graph into a series of snapshot graphs at each moment is a traditional way to tackle evolving graph related problems [7], [8]. As an example, Kim et al. [8] first cluster individual snapshots into quasi-cliques and then map them in adjacent snapshots over time. Inspired by this approach, we design a baseline for cluster evolution tracking, which characterizes the cluster evolution at consecutive moments, by identifying certain heuristic patterns:

- If $\frac{|C_t \cap C_{t+1}|}{|C_t \cup C_{t+1}|} \geq \kappa$ and $|C_t| \leq |C_{t+1}|$, $C_{t+1} = \uparrow C_t$;
- If $\frac{|C_t \cap C_{t+1}|}{|C_t \cup C_{t+1}|} \geq \kappa$ and $|C_t| > |C_{t+1}|$, $C_{t+1} = \downarrow C_t$.

where C_t and C_{t+1} are any two clusters detected at moment t and $t+1$ respectively, $\kappa\%$ is the minimal commonality to say C_t and C_{t+1} are different snapshots of the same cluster. A higher $\kappa\%$ will result in a higher precision but a lower recall of the evolution tracking. Empirically we set $\kappa\% = 90\%$ to guarantee the quality. It is worth noting that this baseline generates the same clusters as the eTrack algorithm, but with a non-incremental evolution tracking approach.

Precision and Recall. To measure the quality of event detection, we use HashtagPeaks, UnigramPeaks and Louvain as baselines to compare with our algorithm eTrack. It is worth noting that Baseline 3 is designed for the tracking of event evolution patterns between moments, so we omit it here. We compare the precision and recall of top 20 events generated by baselines and eTrack and show the results in Table 2(c). Compared with the ground truth, HashtagPeaks and UnigramPeaks have rather low precision and recall scores, because of their poor ability in capturing event bursts. Notice that multiple extracted events may correspond to the same ground truth event. eTrack outperforms the baselines in both precision and recall. Since there are many events discussed in the social media but not very noticeable in news websites, we also validate the precision of the generated events using Google Trends. As we can see, HashtagPeaks and UnigramPeaks perform poorly under Trends validation, since the words they generate are less informative and not very event-indicating. eTrack gains a precision of 95% in Google Trends, where the only failed result is "Samsung galaxy nexus", whose volume is steadily high without obvious peaks in Google Trends. The reason may be that the social stream is very dynamic. Louvain is worse than eTrack. The results show eTrack is significantly better than the baselines in quality.

Life Cycle of Cluster Evolution. Our approach is capable of tracking the whole life cycle of a cluster, from birth to death. We explain this using the example of "CES 2012", a major consumer electronics show held in Las Vegas from January 10 to 13. As early as Jan 6, our approach has already detected some discussions about CES and generated an event about CES. On Jan 8, most people talked about "CES prediction", and on Jan 9, the highlighted topic was "CES tomorrow" and some hearsays about "ultrabook" which would be shown in CES. After the actual event happened on Jan 10, the event grew distinctly bigger, and lots of products, news and messages are spreading over the social network, and this situation continues until Jan 13, which is the last day of CES. Afterwards, the discussions become weaker and continue until Jan 14, when "CES" was not the biggest mention on that day but still existed

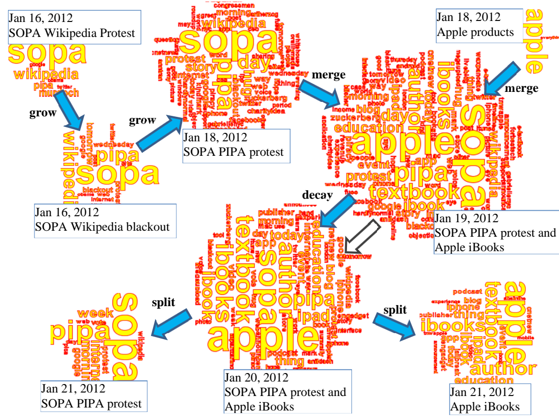
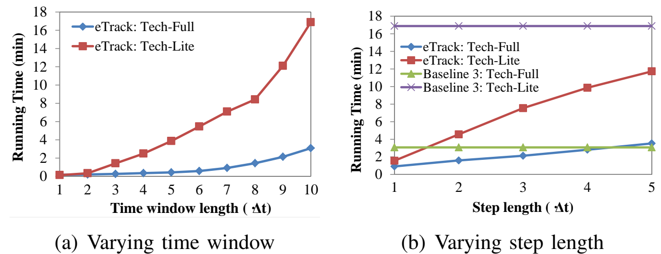


Fig. 9. The merging and splitting of "SOPA" and "Apple". At each moment, an event is annotated by a word cloud. Baselines 1 and 2 only works for the detection of new emerging events, and is not applicable for the tracking of merging and splitting dynamics. The evolution trajectories of eTrack and Baseline 3 are depicted by solid and hollow arrows respectively.



(a) Varying time window

(b) Varying step length

Fig. 10. The running time on two datasets as the adjusting of the time window length and step length.

in some discussions. Compared with our approach, Baselines 1 and 2 can detect the emerging of "CES" with a frequency count at each moment, but no trajectory is generated. Baseline 3 can track a very coarse trajectory of this event, i.e., from Jan 10 to Jan 12. The reason is, if an event changes rapidly and many posts at consecutive moments cannot be associated with each other, Baseline 3 will fail to track the evolution. Since in social streams the posts usually surge quickly, our approach is superior to the baselines.

Cluster Merging & Splitting. Figure 9 illustrates an example of cluster merging and splitting generated by algorithm eTrack. eTrack detected the event of SOPA (Stop Online Piracy Act) and Wikipedia on Jan 16, because on that day Wikipedia announced the blackout on Wednesday (Jan 18) to protest SOPA. This event grew distinctly on Jan 17 and Jan 18, by inducing more people in the social network to discuss about this topic. At the same time, there was another event detected on Jan 18, discussing Apple's products. On Jan 19, actually the SOPA event and Apple event were merged, because Apple joined the SOPA protest and lots of Apple products such as iBooks in education are directly related to SOPA. This event evolved on Jan 20, by adding more discussions about iBooks 2. Apple iBooks 2 was actually unveiled in Jan 21, while this new product gained lots of attention, people who talked about iBooks did not talk about SOPA anymore. Thus, on Jan 21, the SOPA-Apple event was split into two events, which would evolve independently afterwards. Unfortunately, *the above merging and splitting process cannot be tracked by any of the baselines, which output some independent events.*

C. Running Time of Evolution Tracking

Remind that Baseline 1 and 2 are for event identification in a fixed time window. For evolution tracking, we measure how the Baseline 3 and eTrack scale w.r.t. both the varying time window width and the step length. We use both Tech-Lite and Tech-Full streams, and set the time step interval $\Delta t = 1$ day for Tech-Lite, $\Delta t = 1$ hour for Tech-Full to track events on different time granularity. The streaming post rates for Tech-Lite and Tech-Full are 11700/day and 7126/hour respectively. Figure 10(a) shows the running time of eTrack when we increase the time window length, and we can see for a time window of $10\Delta t$ hours in Tech-Full, our approach can finish the post preprocessing, post network construction and event tracking in just 3 minutes. A key observation is that the running time of eTrack does not depend on the overall size of the dataset. Rather, it depends on the streaming speed of posts in Δt. Thus, Tech-Lite takes more time than Tech-Full since its streaming posts in Δt is higher. Figure 10(b) shows if we fix the time window length as $10\Delta t$ and increase the step length of the sliding time window, the running time of eTrack grows nearly linearly. Compared with our incremental computation, Baseline 3 has to process posts in the whole time window from scratch at each moment, so the running time will be steadily high. If the step length is larger than $4\Delta t$ in TechFull, eTrack does not have an advantage in running time compared with Baseline 3, because a large part of post network is updated at each moment. However, this extreme case is rare. Since in a real scenario, the step length is much smaller than the time window length, our approach shows much better efficiency than the baseline approach.

VIII. Related Work

Related work mainly falls in one of these categories.

Clustering and Evolving Graphs. In this paper, we summarize an original post network into a skeletal graph based on density parameters. Compared with partitioning-based approaches (e.g., K-Means [18]) and hierarchical approaches (e.g., BIRCH [18]), density-based clustering (e.g., DBSCAN [18]) is effective in finding arbitrarily-shaped clusters, and is robust to noise. The main challenge is to apply density-based clustering on fast evolving post networks. CluStream [23] is a framework that divides the clustering process into an online component which periodically generates detailed summary statistics for nodes and an offline component which uses only the summary statistics for clustering. However, CluStream is based on K-Means only. DenStream [9] presents a new approach for discovering clusters in an evolving data stream by extending DBSCAN. This work is related to us in that both employ density based clustering. The differences between our approach and DenStream were discussed in detail in the introduction. Subsequently, DStream [19] uses an online component which maps each input data record into a grid and

an offline component which generates grid clusters based on the density. Another related work is by Kim et al. [8], which first clusters individual snapshots into quasi-cliques and then maps them over time by looking at the density of bipartite graphs between quasi-cliques in adjacent snapshots. Although [8] can handle birth/growth/decay/death of clusters, it is not incremental and the split and merge patterns are not supported. In contrast, our approach is totally incremental and is able to track composite behaviors like merging and splitting.

Social Stream Mining. Weng et al. [24] build signals for individual words and apply wavelet analysis on the frequency of words to detect events from Twitter. Twitinfo [2] detects events by keyword peaks and represents an event it discovers from Twitter by a timeline of related tweets. Recently, Agarwal et al. [10] discover events that are unraveling in microblog streams, by modeling events as dense clusters in highly dynamic graphs. Angel et al. [11] study the efficient maintenance of dense subgraphs under streaming edge weight updates. Both [10] and [11] model the social stream as an evolving entity graph, but suffer from the drawback that post attributes like time and author cannot be reflected. Another drawback of [10] and [11] is that they can only handle edge-by-edge updates, and cannot handle subgraph-by-subgraph bulk updates, which are key to efficiency. Both drawbacks are solved in our paper.

Topic/Event/Community detection and tracking. Most previous works detect events by discovering topic bursts from a document stream. Their major techniques are either detecting the frequency peaks of event-indicating phrases over time in a histogram, or monitoring the formation of a cluster from a structure perspective. A feature-pivot clustering is proposed in [5] to detect bursty events from text streams. Sarma et al. [4] design efficient algorithms to discover events from a large graph of dynamic relationships. Jin et al. [25] present Topic Initiator Detection (TID) to automatically find which web document initiated the topic on the Web. Louvain method [22], based on modularity optimization, is the state-of-the-art community detection approach which outperforms others. However, Louvain method is not robust against massive noise, such as is present in Twitter streams. *None of the above works address the event evolution tracking problem.* There is less work on evolution tracking. An event-based characterization of behavioral patterns for communities in temporal interaction graphs is presented in [7]. A framework for tracking short, distinctive phrases (called "memes") that travel relatively intact through on-line text was developed in [1]. The evolution of communities in dynamic social networks is tracked in [26]. Unlike them, we focus on the incremental tracking of cluster evolution in highly dynamic networks.

IX. Conclusion

Our main goal in this paper is to track the event evolution patterns from highly dynamic networks. To that end, we summarize the network by a skeletal graph and monitor the updates to the post network by means of a sliding time window. Then, we design a set of primitive operations and express the cluster evolution patterns using these operations. Unlike previous approaches, our evolution tracking algorithm eTrack performs incremental bulk updates in real time. We deploy our approach on the event evolution tracking task in social streams, and experimentally demonstrate the performance and quality on two real data sets crawled from Twitter. Our experiments show that our approach outperforms the baselines. In the future, it would be interesting to investigate the evolution of social emotions on products for business intelligence.

References

[1] J. Leskovec, L. Backstrom, and J. M. Kleinberg, "Meme-tracking and the dynamics of the news cycle," in *KDD*, 2009, pp. 497–506.

[2] 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 *CHI*, 2011, pp. 227–236.

[3] T. Sakaki, M. Okazaki, and Y. Matsuo, "Earthquake shakes twitter users: real-time event detection by social sensors," in *WWW*, 2010, pp. 851–860.

[4] A. D. Sarma, A. Jain, and C. Yu, "Dynamic relationship and event discovery," in *WSDM*, 2011, pp. 207–216.

[5] G. P. C. Fung, J. X. Yu, P. S. Yu, and H. Lu, "Parameter free bursty events detection in text streams," in *VLDB*, 2005, pp. 181–192.

[6] H. Becker, M. Naaman, and L. Gravano, "Learning similarity metrics for event identification in social media," in *WSDM*, 2010, pp. 291–300.

[7] S. Asur, S. Parthasarathy, and D. Ucar, "An event-based framework for characterizing the evolutionary behavior of interaction graphs," in *KDD*, 2007, pp. 913–921.

[8] M.-S. Kim and J. Han, "A particle-and-density based evolutionary clustering method for dynamic networks," *PVLDB*, vol. 2, no. 1, pp. 622–633, 2009.

[9] F. Cao, M. Ester, W. Qian, and A. Zhou, "Density-based clustering over an evolving data stream with noise," in *SDM*, 2006.

[10] M. K. Agarwal, K. Ramamritham, and M. Bhide, "Real time discovery of dense clusters in highly dynamic graphs: Identifying real world events in highly dynamic environments," *PVLDB*, vol. 5, no. 10, pp. 980–991, 2012.

[11] A. Angel, N. Koudas, N. Sarkas, and D. Srivastava, "Dense subgraph maintenance under streaming edge weight updates for real-time story identification," *PVLDB*, vol. 5, no. 6, pp. 574–585, 2012.

[12] J. Allan, Ed., *Topic detection and tracking: event-based information organization*. Kluwer Academic Publishers, 2002.

[13] Y. Yang, T. Ault, T. Pierce, and C. W. Lattimer, "Improving text categorization methods for event tracking," in *SIGIR*, 2000, pp. 65–72.

[14] C. D. Manning, P. Raghavan, and H. Schuetze, *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[15] D. N. et. al., "A survey of named entity recognition and classification," *Lingvisticae Investigationes*, vol. 30, no. 1, pp. 3–26, 2007.

[16] D. Klein and C. D. Manning, "Accurate unlexicalized parsing," in *ACL*, 2003, pp. 423–430.

[17] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *KDD*, 1996, pp. 226–231.

[18] J. Han, M. Kamber, and J. Pei, *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2011.

[19] Y. Chen and L. Tu, "Density-based clustering for real-time stream data," in *KDD*, 2007, pp. 133–142.

[20] L. W. et.al., "Density-based clustering of data streams at multiple resolutions," *TKDD*, vol. 3, no. 3, 2009.

[21] M. Halvey and M. T. Keane, "An assessment of tag presentation techniques," in *WWW*, 2007, pp. 1313–1314.

[22] V. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *J. Stat. Mech.,*, 2008.

[23] C. C. Aggarwal, J. Han, J. Wang, and P. S. Yu, "A framework for clustering evolving data streams," in *VLDB*, 2003, pp. 81–92.

[24] J. Weng and B.-S. Lee, "Event detection in twitter," in *ICWSM*, 2011.

[25] X. Jin, W. S. Spangler, R. Ma, and J. Han, "Topic initiator detection on the world wide web," in *WWW*, 2010, pp. 481–490.

[26] X. Zhao, A. Sala, C. Wilson, X. Wang, S. Gaito, H. Zheng, and B. Y. Zhao, "Multi-scale dynamics in a massive online social network," in *IMC*, 2012, pp. 171–184.