Keyword Search on microblog Data Streams: Finding Contextual Messages in Real Time

Manoj K Agarwal  
Search Technology Center  
Microsoft India  
agarwalm@microsoft.com

Divyam Bansal  
Bangalore, India  
Google Inc.  
divyamb@google.com

Mridul Garg, Krithi Ramamritham
Dept. of Computer Sc. and Eng.  
IIT-Bombay, India  
gmridul09@gmail.com, krithi@cse.iitb.ac.in

ABSTRACT
Microblogging streams contain information pertaining to emerging real world events. Due to the rapid pace at which these data streams are generated, it is often difficult for users to discover the most relevant messages in the context of their keyword queries. Search over such data streams returns the most recent messages only; most recent messages may not be the most relevant messages. Hence users have to resort to the cumbersome task of sifting through a large amount of information to obtain the context of a live event. We present a novel real time search system – Contextual Event Search – on dynamic message streams, to extract meaningful summaries for live events in real time. Our technique is unsupervised and automatically identifies different facets of the live events in a scalable and effective manner.

We demonstrate that for a given keyword search, users are presented with meaningful, compact and complete contextual event summaries for the most relevant events in a given time window, thus exposing the full context behind the messages.

Categories and Subject Descriptors
H.3.3 [Information Search and Retrieval]: Information filtering, Relevance feedback, Search Process.

General Terms
Algorithms, Experimentation.

Keywords
Dynamic Graph, Indexing, Search, Summarization.

1. INTRODUCTION
Highly dynamic unstructured data streams – sequences of chronologically ordered messages posted by multiple users at a fast pace – occur in various social media and enterprise domains. In microblog streams (e.g., Twitter), messages are posted at a high rate due to their large user base. Twitter is often the first medium to report emerging events [1][2], ranging from globally important events to the events relevant only for a small community.

An event is a real world or an abstract activity, relevant for a group of people or a community. An event in a data stream is defined by “messages, posted by multiple users, in the same context, within a bounded time window”, for example, messages posted by the fans during the course of a football match. It is only natural that in a fast moving world, a large number of events occur concurrently.

Existing unsupervised approaches identify emerging events as clusters of keyword over dynamic message streams [1][2][3]. Each keyword cluster forms an ‘event-topic’. The technique described in [1], when used to discover events from the tweets posted during the Nairobi terrorist attack [7], discovered many event-topics including one containing the keywords:

- A: UK, #kenya, #westgate, #nairobi

Clearly, the context behind the keyword cluster is not available to the users – the keywords are insufficient to describe the underlying event. The same is true of another event-topic:

- B: was, 69, kofi, among, #ghana, attacks, ghanaian, awoonor, killed, poet, prof., #kenya

To better understand the event-topic, i.e., what the event is about, users are needed to search for the most relevant messages in the data stream by themselves. Besides burdening the users with the task of understanding the emerging events manually, for example, to determine if there is connection between A and B, this approach suffers from many shortcomings:

(1) Message search is primitive, e.g., Twitter just returns the most recent tweets for a given search query [5]. It is not necessary that the recent tweets alone are the most relevant tweets for the event.

(2) Simple keyword search results can produce an information overload for a fast moving data stream. Often a large number of tweets are returned by Twitter in response to a search query [6].

(3) In such fast moving data streams, typically the rate at which messages are generated is high but messages are short. Therefore, it is often difficult for the users to understand the context of a standalone message even if the message is informative.

(4) Events evolving in real time comprise different facets. Search results are continuously updated with recent messages and it becomes difficult for the user to keep pace with evolving events.

1.1 Contextual Search on Live Data Streams
We demonstrate our system for Contextual Event Search, to extract the complete contextual summaries for the events unraveling in a live message stream in real-time. The summaries are stored in an event thread as shown in Figure 1. Live real-world events are not just point events – they evolve continuously. The event summary must be updated every time there are significant changes in the event. When these changes are arranged temporally, an event thread results. The event thread captures the passage of time naturally.

Challenges involved in discovering such event threads are many:
The first challenge is to identify and associate the relevant messages to the corresponding ‘event topic’.

Secondly, most real world events do not evolve linearly and comprise several facets. Therefore, the summary for an event is represented as a Directed Acyclic Graph (DAG). It is non-trivial to discover such ‘contextual event threads’. Each unique path in the event thread is a different facet of the event.

The contextual event summary displayed in Figure 1 was constructed by our approach automatically for the event ‘Nairobi terrorist attack’. The event summary starts with a message that a mall has been attacked, followed by how the action against attackers was progressing, rumors, claims and counter claims by authorities and citizens, etc. were also discovered in real time. The important sub-events were discovered from approximately 164K tweets and arranged in a chronological sequence. For each of the sub-events in the event thread, our technique identified an appropriate summary as shown in Figure 1. Sub-event 9 corresponds to A and sub-event 5 corresponds to B.

We demonstrate our system that automatically constructs such summaries as shown in Figure 1, for a live data stream. Our system summarizes the event in a fast moving data stream in real time in an unsupervised manner. The event thread represent a compact, complete and meaningful event summary. A minimal set of related messages are identified that represent the complete event summary. The event summary discovered by our system is stable, i.e., it is updated only with additional information, which is appended to the summary discovered thus far. Our system also exposes the different facets of live events which are presented as a contextual event summary thread. The details of discovering the event threads are beyond the scope of this paper.

To the best of our knowledge, ours is the first system that discovers contextual event threads automatically for a fast moving data stream in real time in an unsupervised manner. Lin et al., explore the problem of generating storylines from microblog data [4]. Their system is only applicable to retrospective data analysis where on relevant tweets a storyline is generated via graph optimization. In [6], Shou et al. present a technique to summarize a twitter data stream, filtered in the context of a given user query. In [9], authors present a method to summarize a pre-specified event topic. Their methodology is applicable for structured and recurring events such as sports events and need the prior knowledge of similar events.

1. Al-Shabaab says it attacked #Westgate mall in #Nairobi to retaliate for Kenya’s role in Somalia.
3. MAJOR assault by security forces ongoing to end two-day siege at Westgate mall. Fears about death toll could be higher when dust settles.
4. KENYA UPDATE: Death toll in #Westgate siege rises to 68 as 9 more bodies recovered during rescue operation.
5. Ghanaian poet & author- Prof. Kofi Awoonor was among the 69 killed in the attack on #Nairobi’s #Westgate mall. #Ghana #Kenya
6. Spread to all Kenyans - the westgate situation may be trying to distract Nairobi, a bigger attack may happen, STAY INDOORS- #SHARE.
7. Israeli forces enter Nairobi mall: security source http://t.co/E0IoM7isPA uc2026 #westgate
8. Two helicopters landed on the roof of #Westgate mall where #Nairobi hostage crisis continues.
9. Speculation that convertive ‘white widow’ Samantha Lewthwaite from UK is the mastermind of the attack on #Westgate Mall in #Nairobi, #Kenya
10. Kenyan forces kill two terrorists, claim control of Westgate mall: Kenyan forces assaulted their ground in Nairobi... http://t.co/zoldHuAaam
11. Something I never saw in 30 yrs as journalist: civilians bringing food, coffee to journalists covering #Nairobi’s #Westgate siege. Amazing!
12. Militants at the Westgate mall in Nairobi, Kenya, are still holding their ground, Somalia’s Al-Shabaab group claims
13. Goosh RT @Lady, Elsie: Hajiy! !’@sirfender: Huh? RT @jstudius: More gunfire, one explosion at #westgate mall. Obviously not over.

1.2 System Components

With the aid the event summaries, we enable the contextual search over data streams. Following are the components of our system:

**Discovery**: The event threads are discovered in a live data stream. They contain the most relevant messages in a chronologically ordered event threads representing its story line. Event thread is associated with a rank based on event popularity and its dynamicity. If an event is highly dynamic with fast updates, its rank increases.

**Indexing**: An index is maintained over the events threads. Since the index is updated in real time, we adopt a lazy-update strategy, i.e., index is updated only for the most popular events. For events, which are less popular, only a subset of these events are updated in the index, unless the underlying changes in the event result in significant increase in event rank.

**Search**: For a keyword query, a ranked list of most relevant event threads is returned. Hence, even the messages which may not contain the query keywords but are part of the event threads are returned. Thus, our system is able to find the most relevant contextual messages for a given keyword query.

The architecture of our system is shown in Figure 2. It contains three components; Event Discovery and Summarization Engine to discover event thread over a live data stream. The discovered events are pushed to the Indexing Engine, which maintains an index over them as well as keeps the index updated for live events. Finally, the Search Engine finds and returns the most relevant topK events, upon receiving a keyword query. topK is a tunable parameter. Event Discovery and Summarization Engine is based on the model in [1] but its details are beyond the scope of this paper. In next section, we present the details of other two components.

**Figure 1. Contextual Event Summary Thread Discovered by our system for Nairobi Attack**

![Diagram of Event Discovery and Summarization Engine](image)

**Figure 2: The system Architecture**
2. REAL TIME CONTEXTUAL SEARCH

Each event is identified by a unique event ID e. As an event evolves, its summary is appended with the most recent updates. Each node in the event tree represents an ‘event topic’ and has a set of relevant tweets associated with it [1]. The algorithm to find meaningful summary of an ‘event topic’ and discovery of different event facets is beyond the scope of this paper. Each event topic, i.e., keyword cluster \( c_{ei} \) has a summary \( s_{ei} \) and a ranking score \( r_{ei} \). For an event \( e_i \), there is a list of clusters \( c_{ei} \), \( i \geq 1 \) associated with it.

Ranking score \( r_{ei} \) for an event \( e \) is computed as \( r_{ei} = \sum_{l=1}^{s_{ei}} c_{ei} \). Event summary \( s_{ei} \) is defined by arranging the cluster summaries \( (s_{ei}, s_{2i}, s_{3i}..) \) in a (multi-faceted) event thread.

2.1 Indexing Engine

The search engine maintains an inverted index. The inverted index consists of words mapped to a posting list of event IDs. For each word \( w \), its posting list \( L_w \) contains event IDs which have \( w \) in its summary. In the posting lists, events are sorted in the decreasing order of their ranking scores. One of the challenges is to maintain the posting lists sorted, since inserting an event in \( L_w \) would take \( O(\log |L_w|) \) time; \( |L_w| \) is the number of events in \( L_w \). To reduce the cost of insert operation, we adopt the following approach: List \( L_w \) is organized as a list \( L_w^{t} \) containing sequence of buckets \( B_1, B_2,.., B_m \) where \( m = \log |L_w| \). Each bucket contains a max-heap and a min-heap. Both heaps contain same event IDs. We define the size of the bucket as the size of its max-heap. Each bucket \( B_i \) has an event with maximum score \( s_{i}^{\text{max}} \) and an event with minimum score \( s_{i}^{\text{min}} \). The sequence of buckets is such that the following property is satisfied:

\[
s_{i-1}^{\text{max}} \geq s_{i}^{\text{max}} \quad \forall i
\]

The size of buckets \( B_i \) increases by a factor of 2 \( (B_{i-1} / |B_i| = 2 \) \( \forall i \leq m \)). Therefore, \( m = \log |L_w| = O(\log(\log |L_w|)) \). The time complexity of the insert operation is equal to the time complexity of a) searching the bucket \( B \) in which the event should be inserted \( O(|L_B|) \); b) inserting the event in \( B \) \( O(|L_B|) \), since the size of each bucket is \( O(\log |L_B|) \) and the bucket is maintained as a heap; c) adjusting the size of \( B \) (its size increases by 1 on inserting an event). Step (c) takes \( O(n^2) \) time since the event with minimum ranking score is removed from \( B \) and inserted in next bucket and this procedure may continue till the first bucket in the sequence. If the size of the last bucket is larger than the maximum allowed, then the event with minimum ranking score is removed and is inserted in a new bucket appended at the end (number of buckets in list \( L_w \) increases by 1).

- The insert operation in each bucket is performed by inserting the event in max-heap and min-heap \( O(|L_B|) \) complexity.
- The remove operation in a bucket is performed by removing the minimum element in its heaps. This is \( O(1) \) for min-heap but for max-heap it takes \( O(L_B) \) if done naively. We store the pointer to the location of the minimum ranking score event in max-heap hence removal takes \( O(n) \) time.

Since there are \( m \) buckets and insertion and removal in each bucket takes \( O(m) \) time, step (c) takes \( O(m^2) \) time. Therefore, the overall insert operation takes \( O(m^2) \) time.

For each list, we maintain a mapping of event ID to the bucket which contains it. This is useful when an event has to be removed from a posting list or its ranking score has to be updated. With this map searching an event in a list takes \( O(1) \) time. Buckets are implemented as locator heap in which a map is maintained which contains the location of event IDs inside the bucket (i.e., heap) thus making the deletion of an event from the bucket \( O(m) \).

2.2 Search Engine

In this section, we describe our query processing system or search engine. The search engine takes a keyword query and returns top \( K \) most relevant event summaries. Suppose a user enters a query \( Q \) of length \( q, \bar{q}, \ldots, q_i \), where \( q || \bar{q} \ldots q || \bar{q} \) is a query word. To define top \( K \) relevant events, we compute \( \text{maxScore}(Q, e) \) [8] of an event \( e \) for a query \( Q \). We define a function \( p(w, e) \) for a word \( w \) and an event \( e \).

\[
p(w, e) = \text{Max} \{ \text{partial_maxScore}(w, e) \}
\]

where \( r_e \) is the event rank. top \( K \) events with highest \( \text{maxScore} \) form the output. We next define some important terms and data structures before describing the algorithm to find top \( K \) events:

- **partial_maxScore**: Initial partial_maxScore of all events is set to \( 0 \). Maximum partial_maxScore of an event \( e \) is the product of \( r_e \) and the words in the query for which \( e \) is present in the posting list of word \( w \), otherwise \( p(w, e) = 0 \).

\[
\text{maxScore} = r_e \times \sum_{l=1}^{q} p(q_j, e)
\]

- **topEvents**: We maintain a min-priority queue of topEvents which stores candidate events and is initially empty. An event \( e_i \) is considered “less than” event \( e_j \) if \( \text{partial_maxScore} \) of \( e_i \) is less than that of \( e_j \).

- **min_topEvents**: It is the event with minimum partial_maxScore among all the events in topEvents.

- **fcEvents**: It stores final candidate events. Any event evicted from topEvents is stored in fcEvents.

- **getMax (B)** returns the event with maximum ranking score in the bucket \( B \).

We search for the posting list for each word in the query. Words for which no posting list is found are discarded. For all the posting lists in consideration, an iterator is set to the first bucket. The highest ranked event from each bucket is inserted in topEvents. We take the event \( \text{min_topEvents} \) and compare it to the event next to it, called \( e_i \), from the posting list it belongs to. We check if \( e_i \) is a candidate event. \( e_i \) is a candidate event if either the number of events in topEvents and fcEvents is less than top \( K \) or \( r_{\text{min_topEvents}} \times \text{partial_maxScore}(Q, e_i) \) which is the maximum possible value of \( \text{partial_maxScore} \) for event \( e_i \). Hence, if this value is less than \( \text{min_topEvents}.\text{partial_maxScore} \) and at least top \( K \) events are already present in topEvents and fcEvents, then \( e_i \) cannot occur in the final output. If it is a candidate event, it is inserted in the topEvents; otherwise the \( \text{min_topEvents} \) may occur in the final output and hence moved from topEvents to fcEvents. If the event \( e_i \) is already present in topEvents or fcEvents, then we just update its \( \text{partial_maxScore} \). Once no more events can be inserted in topEvents, we move the remaining events from topEvents to fcEvents. top \( K \) events with highest \( \text{partial_maxScore} \) in fcEvents form the Output. For faster retrieval, no posting list is traversed beyond \( 2 \times \text{topK} \) event (which lies in \( \log(2 \times \text{topK}) \) bucket).
3. DEMONSTRATION

We demonstrate the ability of our system to find the most relevant messages in the context of a user keyword query. Specifically, for a given keyword query, we demonstrate:

- The ability of our system to find the most relevant messages in real time over live data streams.
- The discovery of contextual tweets in a live data stream, for a given user query, i.e., those tweets that do not even have the query keywords but are relevant.

The ability of our system to create a story line for events unraveling in a live data stream in an unsupervised manner.

Our system returns a ranked list of the most relevant events for the user query. We will demonstrate the statistical summary of the live event including the number of tweets posted for that event and its ranking score. We will also demonstrate that the event summary discovered by our system is complete. At the demo, a user can see a fraction of randomly selected tweets from a live data stream and be able to compare our summary with the raw data. We will demonstrate our system on recorded as well as live Twitter stream.

The screenshot in Figure 3 shows the output of our search system for a given user query. For each event present in the result set, for the given keyword query, users can see a chronologically ordered DAG of most relevant tweets, representing contextual event threads, by clicking on the link.

![Figure 3: Real Time Search Engine over Live Data Streams](image)

4. REFERENCES


