Event Extraction in The Twittersphere

Adel Ardalan[†], Qian Wan[†], Nikesh Garera[‡], AnHai Doan[†‡], Jignesh Patel[†]
[†] University of Wisconsin-Madison
[‡] WalmartLabs
{adel, qwan}@cs.wisc.edu, nikesh@walmartlabs.com, {anhai, jignesh}@cs.wisc.edu

Technical Report

1. INTRODUCTION

Twitter is an online microblogging platform which allows its users to post messages of length up to 140 characters. Users share their opinions, and promote and discuss current events. An event in the Twittersphere happens when many users tweet about a subject (possibly, a real-world event) at a particular time. Table 1 shows some sample events and their attributes.

An *emerging* event is an event such that the number of Twitter users tweeting about it rises to a significant number, over a certain period of time. That is, an event is an emerging one at a particular time if (1) a considerable number of people are discussing it (it is *hot*) and (2) there are considerably more people talking about the event than before (it is emerging).

With the increasing popularity of the Twitter, extracting these events has become appealing for a wide range of applications. These applications include marketing and customer modeling, social studies, political campaigning, among many others. Thus, we are interested in extracting emerging events on the Twittersphere.

A generic event extraction system receives a real-time stream of tweets as input, and generates a stream of event representations which are discussed about on Twitter; we refer to the this mode of operation as *online processing*. In the online processing mode, the system should deal with various stream processing issues, like buffering and archiving the received messages, failure recovery and backup management.

Alternatively, we could think of an *offline processing* mode in which the input is a sequence of timestamped files, each containing the tweets posted during a specific time interval. For example, we might store an hour or a day of tweets in each file. The event extraction system reads each file from the disk, extracts the events discussed in it and returns the corresponding event representations to the end user. In this work, we consider the offline processing mode for the following reasons:

1. There are many different applications working on the Twitter data in a collection of social media analysis systems, one of which is the event extraction system. Other applications include event monitoring, sentiment analysis, etc.

   Hence, there usually exists a gateway module which reads the tweets from the live firehose and stores them in timestamped files in a general record format to be used by other systems. This gateway would also deal with the online processing issues mentioned above, abstracting those measures from all other systems.

2. There is a huge volume of legacy Twitter data which needs to be processed for various applications. This data is already stored on disk as a sequence of timestamped files.

Thus, we define the problem of event extraction in the Twittersphere as follows: *given a sequence of timestamped tweet files, we want to extract emerging events, with high accuracy and low latency.*

Challenges and Solution Ideas There are two major challenges in solving the event extraction problem:

- It is hard to accurately extract events, since a lot of tweets are about trivia or non-events (e.g. horoscope).
- It is hard to develop a scalable event extraction system, as a result of large volume and rate of incoming tweets.

To address the first challenge, our main idea is that when an emerging event happens in the Twittersphere, some phrases suddenly become hot. Not only these phrases become hot, but also phrases related to the same emerging event start co-occurring together significantly more frequently.

Hence, we find these hot phrases and co-occurring phrase pairs using a set of rules which could be tuned by the user. This is *implicit crowdsourcing*, since we are using the crowd messages to extract events.

Next, we use machine learning (in particular, clustering) to find the set of phrases related to the same emerging event. Then, we use editorial rules and machine learning (here, decision tree classifier) to filter out non-interesting events. This way, we are using rules, machine learning and (implicit) crowdsourcing to solve the event extraction problem.

To address the second challenge, our main idea is to leverage the fact that some of the tasks involved in the event extraction process could be executed in parallel. For example, counting the number of phrases in the tweets posted in different time intervals could be executed simultaneously.

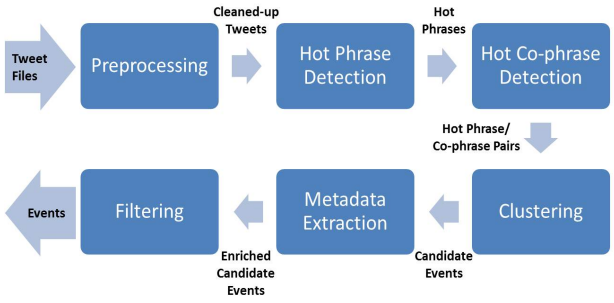

Figure 1: Architecture of Our Event Detection System

Hence, we break the process of extracting emerging events into a pipeline of smaller tasks (like counting phrases). Then, we model the dependency of various tasks in the pipeline. Using this dependency model, we run as many of these tasks as possible in parallel, using a multi-threaded setting.

Our Contributions Our main contributions could be summarized as follows:

- We design and implement a system to solve the event extraction problem, using rules, machine learning and (implicit) crowdsourcing.
- Our implementation is tested against real-world data and shown to be accurate and scalable.

We propose a pipeline of processing modules to detect emerging events, as outlined in Figure 1. From the input tweet file sequence, we first load and preprocess a chunk of English tweets. Each chunk contains a fixed number of English tweets. Preprocessing involves parsing the tweets, performing preliminary filtering and stopword removal, and discarding non-English tweets.

Then, we identify hot and emerging unigram and bigram phrases – called hot phrases for brevity (see Definition 3.1). We count the number of occurrences and the number of users posting each unigram and bigram appearing in the current chunk's tweets, and retain those which satisfy hot and emerging conditions.

Next, using the detected hot phrases in the previous stage, we extract hot and emerging phrase/co-phrase pairs (see Definitions 3.2 and 3.3). We count the number of co-occurrences and number of users posting each pair of phrases in the current chunk's tweets, where the pair contains at least one hot phrase. Then, we only keep the pairs which satisfy the conditions of hot and emerging phrase/co-phrase pairs.

In the next stage, we cluster these pairs based on common phrases, to form event candidates. We then extract a variety of features for each candidate. Next, using these features, we determine whether each candidate represents an interesting event or not, leveraging a set of user-specified rules and a pre-trained decision tree. Finally, the extracted interesting events are presented to the user along with additional information, like metadata and sample tweets.

To achieve scalability, we define the following task types:

1. Chunk Loading (CL)
2. Phrase Counting (PC)
3. Hot Phrase Detection (HP)
4. Co-phrase Counting (CC)
5. Hot Phrase/Co-phrase Pair Detection (HC)
6. Clustering, Metadata Extraction and Filtering (CF)

A task, then, is to run a particular kind of task on a particular chunk, e.g. detecting hot phrases appearing in the 5th chunk.

We then define an order among tasks, based on their data dependency and the chronological order of the chunks they operate on. Next, we use a greedy, dynamic scheduling algorithm to run as many tasks as we can, in parallel. This multi-threaded implementation would achieve a significant speed-up compared to the single-thread implementation.

Our results shows high accuracy (precision of 0.94 and recall of 1.) and low latency (we are able to process an hour worth of tweets in about 1.5 minutes). Many works have considered solving event extraction problem. Examples include [20, 9, 1, 5]. However, to the best of our knowledge, no current piece of work handles the two aforementioned challenges simultaneously.

2. PROBLEM DEFINITION

We start this section by defining how we represent a tweet and an event in our system. Next, we define what an emerging event is. We then proceed to discuss online versus offline event extraction modes. Finally, we define the event extraction problem (in the offline mode).

Definition 2.1 (**Tweet Representation**). A tweet is a message posted by a Twitter user, represented by a record of key-value pairs $[k_1 : v_1, \ldots, k_n : v_n]$. Each key is a string ($k_i \in$ `str`) and each value can be either a (1) string ($v_i \in$ `str`), (2) number ($v_i \in$ `num`), (3) list of values ($v_i \in (r)$) or (4) record of key-value pairs ($v_i \in [k' : v']$). Here, `str` is the set of all strings, `num` is the set of all numbers, (r) is the set of all lists with elements of type r, and $[k' : v']$ is the set of all records of key-value pairs.

Table 2 shows the important fields (keys) of a tweet, their data type and a brief description of what they contain. Most notable among them is the one corresponding to the key TweetText. This value is the textual message of the tweet.

Definition 2.2 (**Event Representation**). We represent an event by a tuple $\langle S, t, D \rangle$, where:

- $S \subset$ `str`, referred to as *event signature*, is a set of strings related to various aspects of the event (like actions, entities, places and time),
- $t \in$ `num` is the the time this event is extracted at, and
- $D \subset [k : v]$ is a set of tweets about the event.

In this study, we want to extract *emerging, dynamic events* (denoted as emerging event for brevity). Intuitively, an event is an emerging one at a particular time if:

1. a considerable number of people are discussing it (it is hot) and

Event Description	Time	Place
Top boss of a Mexican drug cartel was killed in a shootout.	October 09, 2012	Progreso, Mexico
Earthquake happened in Japan.	2:46PM 3/11/2011	Japan's northeastern coast

Table 1: Sample Events

Tweet Field Name	Field Type	Description
TweetText	String	Set to the original tweet text if this is a retweet.
UserID	Integer	ID of the user who has posted this tweet.
Timestamp	Integer	Time when the tweet has been posted (as the number of milliseconds since January 1, 1970, 00:00:00 GMT).
IsReply?	Boolean	Whether this is a tweet in reply to another tweet. Determined by checking whether the first character of the tweet text is "@"
URL	String	The final URL, if any, in the tweet text (translated from the shortened URL, like `t.co/abcd`).
IsRetweet?	Boolean	Whether this is a retweet of another tweet. Determined by checking whether the field corresponding to the retweet text is filled or not.

Table 2: Extracted Fields from Each Tweet

2. there are considerably more people talking about the event than before (it is emerging).

Formalizing the above two conditions, we define an emerging event as follows.

Definition 2.3 (**Emerging Event**). Let $e = \langle S, t, D \rangle$ be an event discussed on the Twitter, $f(e, t_1, t_2)$ be a function which returns the number of tweets in which e has been discussed during the time interval $[t_1, t_2]$ and $u(e, t_1, t_2)$ be a function which returns the number of users who have discussed e in the interval $[t_1, t_2]$. Then, e is an emerging event if:

$$\begin{aligned} f(e, t_h, t) &\geq \mu_f \\ \frac{f(e, t_h, t)}{\frac{1}{h}\sum_{i=1}^{h} f(e, t_{i-1}, t_i)} &\geq \lambda_f \\ u(e, t_h, t) &\geq \mu_u \\ \frac{u(e, t_h, t)}{\frac{1}{h}\sum_{i=1}^{h} u(e, t_{i-1}, t_i)} &\geq \lambda_u \end{aligned}$$

for

$$t_0 < t_1 < \cdots < t_h < t.$$

where

- μ_f is the minimum number of tweets discussing a hot event,
- λ_f is the minimum jump in the number of tweets discussing an emerging event, compared to the average number of tweets posted about that event over some $h > 0$ previous epochs,
- μ_u is the minimum number of people discussing a hot event, and
- λ is the minimum jump in the number of users discussing an emerging event, compared to the average number of people talking about it over h previous epochs.

We want to extract such emerging events from a collection of tweets. We might access these tweets through real-time stream or a collection of files containing archives of tweets posted during certain times periods. Next, we review these two access modes and our choice of which access method to consider.

Online and Offline Event Extraction: A generic event extraction system receives a real-time stream of tweets as input, and generates a stream of event representations which are discussed about on Twitter; we refer the this mode of operation as *online processing*. In the online processing mode, the system should deal with various online processing issues, like buffering and archiving the received messages, failure recovery and backup management.

Alternatively, we could think of an *offline processing* mode in which the input is a sequence of timestamped files, each containing the tweets posted during a specific time interval. For example, we might store an hour or a day of tweets in each file. The event extraction system reads each file from the disk, extracts the events discussed in it and returns the corresponding event representations to the end user.

In this work, we consider the offline processing mode for the following reasons:

1. There are many different applications working on the Twitter data in a collection of social media analysis systems, one of which is the event extraction system. Other applications include event monitoring, sentiment analysis, etc.

   Hence, there usually exists a gateway module which reads the tweets from the live firehose and stores them in timestamped files in a general record format to be used by other systems. This gateway would also deal with the online processing issues mentioned above, abstracting those measures from all other systems.

2. There is a huge volume of legacy Twitter data which needs to be processed for various applications. This data is already stored on disk as a sequence of timestamped files.

Given the above definitions and discussion, we define the event extraction problem as follows.

Definition 2.4 (Twitter Event Extraction Problem). Given:

- a finite sequence of input tweet files $F = (\langle f_i, \tau_i \rangle)_{i \in \mathbb{I}_n}$ where each f_i contains the tweets posted during the time interval $[\tau_{i-1}, \tau_i]$ and $\mathbb{I}_n = \{1, \cdots, n\}$,
- the starting time of the first file τ_0, and
- a set of user-provided parameters Θ,

extract the set $E = \{e_1, \cdots, e_p\}$ where each $e_j = \langle S_j, t_j, D_j \rangle, j \in \mathbb{I}_p$ is the representation of an emerging event extracted from the tweets posted during $[\tau_0, \tau_n]$, to:

- maximize accuracy of the extracted events,
- minimize latency of returning the results to the end user, and
- minimize the total processing time.

The measure of accuracy we use here is the F1 score, the harmonic mean of precision and recall. We define the precision to be the ratio of extracted events which represent an emerging event happening on the Twitter during the target time period. We define the recall to be the ratio of the emerging events happening during the target time period which we have been able to extract a representation for.

We define the latency as the average difference between the time of submission of a new chunk to be processed and the time when the system returns the results for that chunk to the user. Finally, we define the total processing time to be the total time spent processing F. We will discuss the measures we wish to optimize and how we approach the optimization problem in more detail in the following sections.

One of the major issues we address in this work is *scalability*. In the online mode of processing, the scalability is defined as follows: if the rate of input data and amount of available computing resources (memory and processing units) would increase linearly, then the latency and the total processing time of the system would increase with a linear rate as well. This is particularly important observing the increasing trend of Twitter data volume over the past few years. For the offline mode, the scalability could be restated as *processing the data and extracting the events as fast as possible.*

Now, we describe our proposed solution to solve the Twitter event detection problem, to achieve high accuracy and scalability.

3. OUR SOLUTION

One of the main idea behind our solution is to use Twitter as an *implicit crowdsourcing* medium, to extract emerging events. We observe that when an emerging event happens in the Twittersphere, some phrases suddenly become hot. Not only these phrases become hot, but also phrases related to the same emerging event start co-occurring together significantly more frequently.

So, we find these hot phrases and co-occurring phrase pairs, using a set of rules which could be tuned by the user. Next, we use machine learning (in particular, clustering) to find the set of phrases related to the same emerging event. Then, we use editorial rules and machine learning (here, decision tree classifier) to filter out events that are not interesting for the user.

To implement the above idea in a scalable way, we leverage the fact that some of the tasks involved in the event extraction process could be executed in parallel. For example, counting the number of phrases in the tweets posted in different time intervals could be executed simultaneously.

Hence, we break the process of extracting emerging events into a pipeline of smaller tasks (like counting phrases). Then, we model the dependency of various tasks in the pipeline and run as many of these tasks as possible in parallel, using a multi-threaded setup.

3.1 Overview

We propose a pipeline of processing modules to detect emerging events, as outlined in Figure 1. From the input tweet file sequence, we first load and preprocess a chunk of English tweets. Each chunk contains a fixed number of English tweets. Preprocessing involves parsing the tweets, performing preliminary filtering and stopword removal, and discarding non-English tweets.

Then, we identify hot and emerging unigram and bigram phrases – called hot phrases for brevity (see Definition 3.1). We count the number of occurrences and the number of users posting each unigram and bigram appearing in the current chunk's tweets, and retain those which satisfy hot and emerging conditions.

Next, using the detected hot phrases in the previous stage, we extract hot and emerging phrase/co-phrase pairs (see Definitions 3.2 and 3.3). We count the number of co-occurrences and number of users posting each pair of phrases in the current chunk's tweets, where the pair contains at least one hot phrase. Then, we only keep the pairs which satisfy the conditions of hot and emerging phrase/co-phrase pairs.

In the next stage, we cluster these pairs based on common phrases, to form event candidates. We then extract a variety of features for each candidate. Next, using these features, we determine whether each candidate represents an interesting event or not, leveraging a set of user-specified rules and a pre-trained decision tree. Finally, the extracted interesting events are presented to the user along with additional information, like metadata and sample tweets. In the following subsections, we discuss each step in more detail.

3.2 Preprocessing

The main purpose of this stage is to chop the stream (here, the sequence of files) into blocks of clean English tweets, called *chunks*. Each chunk contains the same fixed number of English tweets. Fixed-size chunks would make the performance of the system more predictable, since the rate of incoming tweet stream is not constant. This way, we could assume the processing time of all the chunks are roughly the same. It also makes configuring the system easier.

We load the next chunk of English tweets from the current input tweet file on disk by first reading the next file block to a memory buffer. Then we parse and clean each tweet

in the buffer and tokenize the tweet text. We then discard non-English tweets and create an in-memory collection of English tweet records to be used in the following steps. Next, we describe these steps, discuss the data elements of a tweet record, why we need any of these fields and how we fill in these values using a tweet.

The input to the preprocessing phase consists of:

- the path to the input tweet files on disk,
- the size of a chunk w (in English tweet records),
- a list of English words (we use the the `dict` file shipped with standard Linux distributions),
- an English language classification threshold,
- the collection of blacklisted phrases, and
- the collection of news agency names/keywords.

The output of this stage is the in-memory collection of the current chunk's tweet records C_k, and a timestamp $\hat{\tau}_k$ of the last tweet posted in the current chunk. C_k is an associative array (map) of key-value pairs $(y : v)$ with $y \in$ `num` and v being a tweet record . For each tweet record, we apply a hash function[1] $\hbar$ to the value of "`TweetText`" field (see Definition **??**) and the resulting number will be used as the key to refer to the tweet record in C_k.

Loading and Parsing Tweets: First, we initialize C_k to an empty map, and $\hat{\tau}_k$ to 0. Then, starting from the first block of the first tweet file, we load the next block of tweets into a memory buffer. Each line in the input tweet file corresponds to one tweet. We parse the tweet (from JSON or tab-separated format) and extract the required fields which are described in Table 2.

Discarding Tweets with Blacklisted Phrases: Next, if any substring of the TweetText is in the collection of blacklisted phrases, we discard the tweet and read the next one. Using this feature, the user could indicate the phrases and tweets containing those phrases which she believes would not correspond to any interesting events. For example, we consider any of the zodiac sign names, like "*capricorn*", as a blacklisted phrase.

Detecting Retweets and Updating Retweeting Users: Twitter users use retweeting – reposting a tweet to one's followers – to share interesting messages. Tracking the users retweeting a particular tweet could give us useful information about events. For example, lots of retweets of a few tweets about an event would indicate that it is not very controversial; almost everyone would just pass the messages around without commenting. On the other hand, a very few number of retweets related to an event might indicate that users potentially have many different ideas and opinions about the event.

We check whether the current tweet is a retweet and whether the original tweet has been added to C_k already: we apply $\hbar$ to the TweetText and look it up in C_k. If C_k contains the hash value, then we retrieve the original tweet record from C_k and add the UserID to the set corresponding to the "`UserIDs`" key - the collection of users retweeting the original tweet. We then read the next tweet from the buffer.

On the other hand, if the current tweet is not a retweet, or if it is a retweet but C_k does not contain the original tweet (that is , the original tweet or any retweets of it has not been posted in the current chunk up to this point), we then add a new tweet record to C_k as follows. We first *clean* the TweetText by removing symbols, punctuation marks, letter repetitions and URLs from it. Then, we tokenize the cleaned TweetText with white space delimiter.

Language Classification: Next, we use a dictionary-based English language classifier to determine whether the tweet is in English or not. The classifier counts the number of TweetText tokens contained in the English dictionary and computes the ratio of this number to the total number of tokens. It classifies the tweet as English if this ratio is larger than a particular threshold (0.8 in our experiments).

If the tweet is classified as English, then we instantiate a new tweet record using the fields extracted before and add it to C_k, with the key being the result of applying $\hbar$ to the TweetText field. Also, if the value of Timestamp field of the current tweet is greater than $\hat{\tau}_k$, then we set $\hat{\tau}_k$ to the the value of Timestamp field of the current tweet.

Forming A Chunk: We continue reading input tweets until the current chunk is complete, i.e. either we have added w English tweet records to C_k or we have reached the end of the last tweet file. Finally, we return the current chunk's in-memory collection of tweet records C_k and the timestamp $\hat{\tau}_k$. Figure 2 shows a sample tweet, the extracted fields and the corresponding tweet record after being preprocessed.

Tweet Record and Extracted Fields: We now discuss the information we keep from each English tweet in a tweet record as we form the chunks.

Tweet Record Type: The tweet record type is defined as follows:

```
[
  "TweetText" :          str,
  "Tokens" :           (str),
  "UserIDs" :          {num},
  "IsReply" :        boolean,
  "ContainsNewsKW" : boolean,
  "URL" :              str
]
```

A *tweet record* is an instance of the tweet record type, representing a (re)tweet in the current chunk, as an in-memory associative array (map). Table 3 shows how we fill in a new tweet record's fields using the corresponding tweet (Table 2).

As we mentioned earlier, when we encounter a retweet (decided using the IsRetweet? field) the original tweet record of which we have added to C_k before, we retrieve the original tweet record from C_k and add the UserID to the set corresponding to the "`UserIDs`" key; the rest of the mapping described above has been established for the original tweet and we may save repeating the processing effort.

[1]Here, we use the hash function $\hbar(s) = \sum_{i=0}^{\varsigma-1} s[i].31^{\varsigma-1-i}$ where $s[i]$ denotes the i^{th} character of the string, ς is the length of s, and terms are summed using 32-bit integer addition.

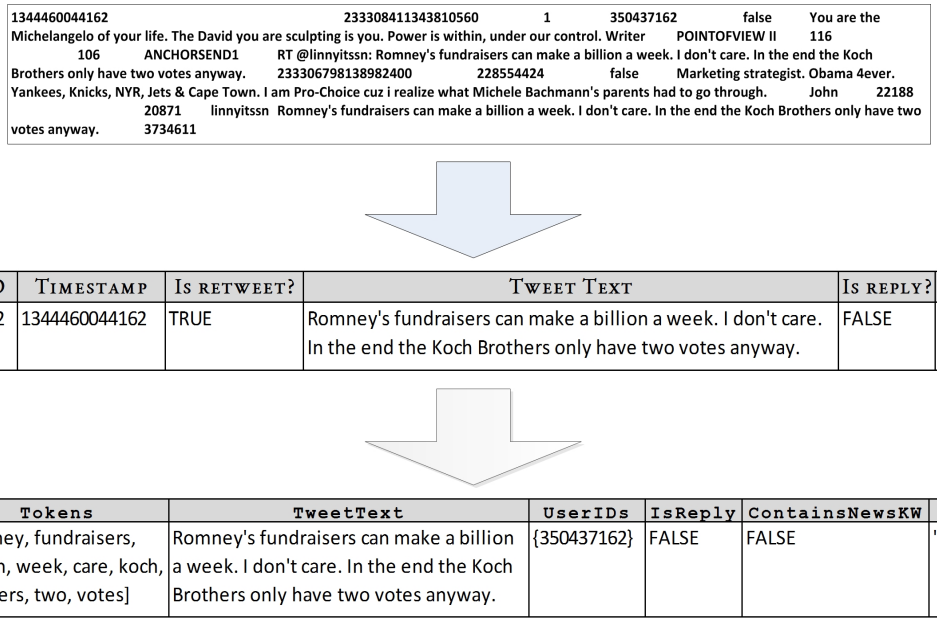

Figure 2: Preprocessing Example

Tweet Record Key	Associated Value	Mapping to Tweet Fields When First Instantiated
"TweetText"	Text of the original tweet (whenever a tweet or its retweet appears in a chunk for the first time, we create a tweet record for the original tweet and for each retweet, add the ID of the user posting the retweet to the set of the users (re)posting the tweet)	Value of TweetText after removing any URLs
"Tokens"	The list of tokens of original tweet's cleaned text	the tokens extracted from the cleaned TweetText
"UserIDs"	The set of user IDs (re)posting this tweet in the current chunk	a set containing the ID of the user posting the tweet (UserID)
"IsReply"	A boolean field indicating whether this tweet is in reply to another tweet	The corresponding value extracted from the tweet (IsReply?)
"ContainsNewsKW"	A boolean field which indicates whether any news agency names/keywords are mentioned in the tweet text	Determined by searching the TweetText for any substring which is contained in the list of news agency keywords/names
"URL"	The final URL (translated from the shortened URLs like t.co/abcd) mentioned in the tweet text (if any)	the URL extracted from the tweet (URL)

Table 3: Tweet Record Fields

The "`TweetText`" field is used to refer to the tweet record and to extract "`Tokens`" after cleaning. The value of "`Tokens`" is in turn used to generate unigrams and bigrams. We store the tokens since cleaning and tokenization are expensive operations and we want to avoid repeating them as much as possible.

The "`UserIDs`" field tracks the users who have (re)tweeted the message. It is used later to figure out how much discussion users have about a candidate event; the more the number of retweets, the less the controversy about the event. The value of the "`IsReply`" field is another indicator of the variety of ideas about an event. People use replies to post reflections about each others' opinions and the more replies posted about an event, the event evokes more discussion.

We extract "`ContainsNewsKW`" to see whether people have potentially referred to an actual news article in their tweet. Since news articles are a more curated and sanity-checked form of information about events, existence of a news article about an event is an indicator of its potential interestingness. The "`URL`" field also is an indicator of outside reference which shows that additional material exists about the event.

Discussion: The output of this step is an associative map C_k consisting of the tweet records extracted from the latest chunk of tweets read from the input stream (here, the currently open tweet file). This map is retained in memory for the lifetime of this chunk. Also, processing later chunks depends on the availability of C_k; particularly, the next h chunks $C_{k+1}, \ldots, C_{k+h}$ need the tweet records in C_k for hot co-phrase detection, as we will discuss later.

Thus, efficient access methods and storage mechanisms are utilized to minimize the overhead of lookups and retrievals, and to reduce the memory needed to store it. These strategies consist of using appropriate associative map implementation to improve the access time, use more primitive types (like long integers for user IDs) whenever possible and having one copy of each string literal.

In the course of developing our system, we have decided to go over the input tweet files F once, divide them into a sequence of preprocessed chunks $(C_i)_{i=1}^{m}$ and store the tweets back on disk. This is also a one-time task which could be done when the tweets are read from the stream. Next, we detect the hot phrases appearing in the current chunk.

3.3 Hot Phrase Detection

The main purpose of this step is to find phrases in the current chunk which are hot (have appeared in a lot of tweets and mentioned by many users) and emerging (their appearance has increased considerably, compared to previous chunks, both in number of appearances and number of users mentioning them). We call such a phrase a *hot phrase* and define it as follows.

Definition 3.1 (**Hot Phrase**). Let $f(w,k) : \texttt{str} \times \mathbb{Z} \mapsto \mathbb{Z}$ be a function that returns the total number of appearances of phrase w in tweet records of C_k, and $u(w,k) : \texttt{str} \times \mathbb{Z} \mapsto \mathbb{Z}$ be a function that returns the total number of users mentioning phrase w in their tweets in C_k. Given the parameters $h, \mu_f, \lambda_f, \mu_u$ and λ_u, w is *a hot phrase in* C_k if:

1. $f(w,k) \geq \mu_f$

2. $\frac{f(w,k)}{\frac{1}{h}\sum_{i=1}^{h} f(w,k-i)} \geq \lambda_f$

3. $u(w,k) \geq \mu_u$

4. $\frac{u(w,k)}{\frac{1}{h}\sum_{i=1}^{h} u(w,k-i)} \geq \lambda_u$

In the hot phrase detection phase, we first generate tweet record phrases (unigrams and bigrams) from the "`Tokens`" field value of the tweet records in C_k. Then for each phrase of C_k tweet records, we count the number of its appearances, store the user IDs of the users posting it, store the C_k tweet records containing it, and determine whether it is hot in C_k based on the criteria described in Definition 3.1.

The inputs to this phase are the map C_k, the list of stopwords and the parameters $h, \mu_f, \lambda_f, \mu_u$ and λ_u. The output of this phase consists of the following:

- An associative map $\mathcal{PQ}_k$ of key-value pairs $(y : v)$ with $y \in$ `str` and $v \in$ `num`. We apply $\hbar$ to each phrase and use it as the key to refer to the phrase's number of appearances in C_k tweet records.
- An associative map $\mathcal{PU}_k$ of key-value pairs $(y : v)$ with $y \in$ `str` and $v \subset$ `num`. We apply $\hbar$ to each phrase and use it as the key to refer to the set of user IDs posting the phrase in C_k tweet records.
- An associative map $\mathcal{PW}_k$ of key-value pairs $(y : v)$ with $y \in$ `str` and $v \subset$ `num`. We apply $\hbar$ to each phrase and use it as the key to refer to the set of C_k tweet records in which the phrase has appeared. Each number in this set is the result of applying $\hbar$ to a tweet's string and the corresponding tweet record could be retrieved by querying C_k with this number.
- A set $\mathcal{H}_k \subset$ `str` of hot phrases in C_k's tweet record phrases.

Phrase Generation: First, we initialize $\mathcal{PU}_k$, $\mathcal{PQ}_k$ and $\mathcal{PW}_k$ to empty maps and $\mathcal{H}_k$ to an empty set. Then, for each tweet record in C_k, we generate unigrams and bigrams using the value of the "`Tokens`" field. We go over the elements of the "`Tokens`" field array and return each token (unigram) as well as the concatenation of every two consecutive tokens (bigram).

Removing the Stopwords and Updating the Maps: For each phrase in the current tweet record's unigrams and bigrams, if the phrase is a stopword or contains a stopword token, then we ignore it and proceed to the next phrase. Some examples of stopwords are "*I*", "*the*", "*LOL*" and "*Good morning*". Considering both unigrams and bigrams as phrases gives us the opportunity to discard bigram stopwords as well.

If the phrase is not a stopword and does not contain a stopword token, then we check whether $\mathcal{PQ}_k$ (as well as $\mathcal{PU}_k$ and $\mathcal{PW}_k$) contains the key corresponding to the current phrase. If not, we add it to $\mathcal{PQ}_k$, $\mathcal{PU}_k$ and $\mathcal{PW}_k$ with the corresponding values 0, $\emptyset$ and $\emptyset$ respectively.

Next, we retrieve the set of user IDs (re)tweeting the current tweet record, stored in the "`UserIDs`" field value. Each user with her ID in this set has posted or reposted a tweet containing the current phrase, so we increment the number of appearances of the phrase in $\mathcal{PQ}_k$ by the size of this set (number of users mentioning the phrase in a tweet), add the user IDs in this set to the corresponding set in $\mathcal{PU}_k$ and add the tweet hash value to the corresponding set in $\mathcal{PW}_k$.

We observe that since each user might post more than one tweet in a chunk and a tweet record might have a phrase w appeared in it more than once, the values of $\mathcal{PQ}_k(w)$ and $\|\mathcal{PU}_k(w)\|$ are different in general. As an example, suppose there are only two tweets "`a b a`" and "`b a c`" in the current chunk, posted by the same user. Then, $\mathcal{PQ}_k(\text{“}\texttt{a}\text{”}) = 3$ whereas $\|\mathcal{PU}_k(\text{“}\texttt{a}\text{”})\| = 1$.

Detecting Hot Phrases: For the chunks where there is not enough history (i.e. $k \leq h$), we simply ignore the rest of the pipeline in order for the results to be consistent with our definitions.

So next, if $k > h$, then for each phrase in $\mathcal{PQ}_k$'s key set, we need the corresponding values in $\mathcal{PQ}_{k-i}$s and $\mathcal{PU}_{k-i}$s for $i = 1, \ldots, h$. If the pipeline is executed sequentially on the sequence of chunks one after another, all of these maps are already filled at this point, since we have completed processing the previous chunks before. Hence, we need to keep maps for the previous h chunks in memory so we prevent repeating the counting procedure.

Now, using Definition 3.1, for each phrase w in $\mathcal{PQ}_k$'s key set, we retrieve the values $f(w, k-i)$ and $u(w, k-i)$ for $i = 0, 1, \ldots, h$ with:

$$f(w, k-i) = \mathcal{PQ}_{k-i}(w)$$
$$u(w, k-i) = \|\mathcal{PU}_{k-i}(w)\|$$

Then, we apply Definition 3.1 to check whether w is a hot phrase; if so, then we add it to the set of hot phrases $\mathcal{H}_k$.

Finally, we return the current chunk's maps $\mathcal{PQ}_k$, $\mathcal{PU}_k$ and $\mathcal{PW}_k$, and the set of hot phrases in the current chunk $\mathcal{H}_k$ to be used in the subsequent steps. Figure 3 illustrates an example of the hot phrase detection process, illustrated using $\mathcal{PQ}$s only.

3.4 Hot Co-phrase Detection

The signature of an event is a collection of terms and phrases related to that event which describe various aspects of it. Not only these phrases would appear more frequently and be posted by more users as an event happens, but the phrases related to a single event tend to *co-occur* in individual tweets more frequently as well. This is due to the fact that users usually mention multiple aspects of an event in their tweets, like who does what and where.

In this step, we detect hot and emerging co-occurrence of hot phrases (detected in the previous step) with other phrases in tweets by scanning the tweet records again and tracing statistics of phrase pairs which have a hot phrase leg. We then pick the ones that show significant appearance and emergence as hot phrase/co-phrase pairs which are used later to form candidate events.

Let's define the notion of a co-phrase.

Definition 3.2 (Co-Phrase). Given a string $W \in$ `str` and the set B_W of W's phrases, the *co-phrase relation* is defined as $B_W \times B_W$, which is an equivalence relation. Particularly, for each pair of phrases $(w_1, w_2) \in B_W \times B_W$, we say that w_2 is a *co-phrase* of w_1.

We define the notion of hot co-phrase as follows.

Definition 3.3 (Hot Co-Phrase). Let $cof(w_1, w_2, k)$: `str` $\times$ `str` $\times \mathbb{Z} \mapsto \mathbb{Z}$ be a function that returns the number

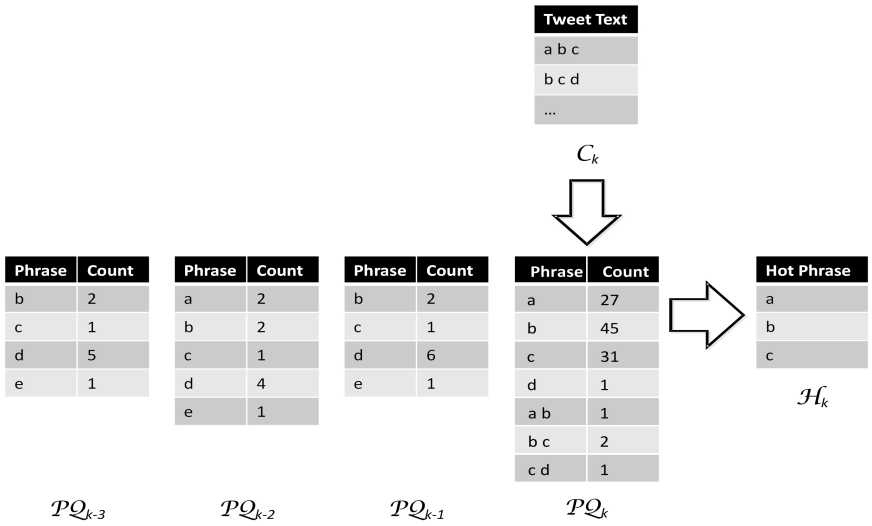

Figure 3: Hot Phrase Detection Example ($\mu_f = 20$, $\lambda_f = 5$ and $h = 3$)

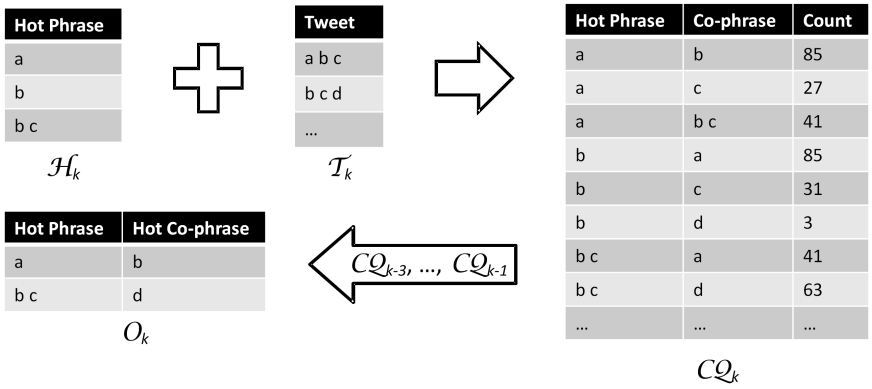

Figure 4: Hot Co-phrase Detection Example

of appearences of phrases w_1 and w_2 together in one of the tweet records in C_k and $cou(w_1, w_2, k) : \texttt{str} \times \texttt{str} \times \mathbb{Z} \mapsto \mathbb{Z}$ be a function that returns the total number of users mentioning phrases w_1 and w_2 together in one of the tweet records in C_k. Given the parameters $h, \mu'_f, \lambda'_f, \mu'_u$ and λ'_u, (w_1, w_2) is *a hot phrase/co-phrase pair in* C_k if:

1. $cof(w_1, w_2, k) \geq \mu'_f$
2. $\frac{cof(w_1,w_2,k)}{\frac{1}{h}\sum_{i=1}^{h} cof(w_1,w_2,k-i)} \geq \lambda'_f$
3. $cou(w_1, w_2, k) \geq \mu'_u$
4. $\frac{cou(w_1,w_2,k)}{\frac{1}{h}\sum_{i=1}^{h} cou(w_1,w_2,k-i)} \geq \lambda'_u$

The goal of this step is to separately count the number of appearances of each phrase/co-phrase pair in the current chunk tweet records C_k and history chunks tweet records $C_{k-h}, \ldots, C_{k-1}$'s phrase/co-phrase pairs, store the user IDs of the users posting each phrase/co-phrase pair in the current chunk and history chunks tweet record phrase/co-phrase pairs, and determine the hot phrase/co-phrase pairs in C_k. The inputs to this step are $C_{k-h}, \ldots, C_{k-1}, C_k$, the set of hot phrases in the current chunk $\mathcal{H}_k$, the list of stopwords and the parameters $h, \mu'_f, \lambda'_f, \mu'_u$ and λ'_u. It returns the set of hot phrase/co-phrase pairs in the current chunk, $\mathcal{O}_k$.

Similar to the hot phrase detection step, we keep track of the C_k's tweet record phrase/co-phrase pairs statistics by instantiating and filling the following data structure:

- the two-dimensional associative map $\mathcal{CQ}_k$ of key-value pairs $((y_1, y_2) : v)$ with $y_1, y_2 \in \texttt{str}$ and $v \in \texttt{num}$ - we apply $\hbar$ to each hot phrase y_1, to the string of y_2 – a co-phrase of y_1 – and use the results to refer to the phrase/co-phrase pair's number of appearances in C_k tweet record phrase/co-phrase pairs,
- the two-dimensional associative map $\mathcal{CU}_k$ of key-value pairs $((k_1, k_2) : v)$ with $k_1, k_2 \in \texttt{str}$ and $v \subset \texttt{num}$ - we apply $\hbar$ to each hot phrase string y_1, to the string of y_2 – a co-phrase of y_1 – and use the resulting pair of numbers to refer to the phrase/co-phrase pair's set of user IDs posting them in C_k tweet record phrase/co-phrase pairs, and
- the set $\mathcal{O}_k \subset \texttt{str} \times \texttt{str}$ of hot phrase/co-phrase pairs in C_k's tweet record phrase/co-phrase pairs.

Initialization: First, we initialize $\mathcal{CQ}_k, \mathcal{CQ}_{k-1}, \ldots, \mathcal{CQ}_{k-h}$ and $\mathcal{CU}_k, \mathcal{CU}_{k-1}, \ldots, \mathcal{CU}_{k-h}$ to empty maps, and the set

$\mathcal{O}_k$ to an empty set. We shall emphasize that for each tweet record chunk C_k, we recalculate $\mathcal{CQ}_{k-1}, \ldots, \mathcal{CQ}_{k-h}$ and $\mathcal{CU}_{k-1}, \ldots,$
$\mathcal{CU}_{k-h}$ for the following reasons:

1. The space required to store the statistics for every pair of phrases is quadratic in the number of C_k phrases, which is not manageable in terms of memory space.

2. Keeping $\mathcal{CQ}_i$ and $\mathcal{CU}_i$ in memory will not be necessarily helpful for detecting hot phrase/co-phrase pairs of subsequent chunks as the set of hot phrases are different for various chunks. Thus, for any hot phrase $w \in \mathcal{H}_{i+j}, j \in \{1, \ldots, h\}$ which $w \notin \mathcal{H}_i$, we still need to retraverse the whole C_i to fill in the entries corresponding to the key $(w, .)$ in $\mathcal{CQ}_i$ and $\mathcal{CU}_i$, needed for computing $\mathcal{O}_{i+j}$.

Phrase/Co-Phrase Pair Generation and Map Updating: Next, we fill in $\mathcal{CQ}_{k-i}$ and $\mathcal{CU}_{k-i}$ maps for all $i = 0, 1, \ldots, h$. For each $i \in \{0, 1, \ldots, h\}$, we go over all the tweet records in C_{k-i} and generate unigrams and bigrams using the value of the "`Tokens`" field. Now, for each hot phrase $w_1 \in \mathcal{H}_k$, if the generated set of phrases of the current tweet record contains w_1, then we iterate w_2 over these phrases in an inner loop. For each phrase w_2 in the current tweet record phrases, if $\mathcal{CQ}_{k-i}$ and $\mathcal{CU}_{k-i}$ do not contain (w_1, w_2), we add the key (w_1, w_2) to $\mathcal{CQ}_{k-i}$ and $\mathcal{CU}_{k-i}$, initializing the corresponding values to 0 and $\emptyset$ respectively.

Next, we retrieve the set of user IDs (re)tweeting the current tweet record which corresponds to the value of the "`UserIDs`" field. Each user with her ID in this set has posted or reposted a tweet corresponding to a tweet record in C_{k-i} containing w_1 and w_2 simultaneously, so we increment the number of appearances of the phrase/co-phrase pair (w_1, w_2) in $\mathcal{CQ}_{k-i}$ by the size of this set (number of users mentioning the phrase/co-phrase pair in a tweet) and add the user IDs in this set to the corresponding set in $\mathcal{CU}_{k-i}$.

Hot Phrase/Co-Phrase Pair Detection: Next, to distinguish the hot phrase/co-phrase pairs based on Definition 3.3, we iterate over the phrase/co-phrase pairs in $\mathcal{CQ}_k$ key set and for each phrase/co-phrase pair (w_1, w_2), retrieve the values $cof(w_1, w_2, k-i)$ and $cou(w_1, w_2, k-i)$ for $i = 0, 1, \ldots, h$ where:

$$cof(w_1, w_2, k-i) = \mathcal{CQ}_{k-i}(w_1, w_2)$$
$$cou(w_1, w_2, k-i) = \|\mathcal{CU}_{k-i}(w_1, w_2)\|$$

If $\mathcal{CQ}_{k-i}$ or $\mathcal{CU}_{k-i}$ do not contain the key (w_1, w_2), we consider the default of 0.

Then, we apply Definition 3.3 to check whether the phrase/co-phrase pair (w_1, w_2) is hot; if it is, we add it to the set of hot phrase/co-phrase pairs $\mathcal{O}_k$. Finally, we return $\mathcal{O}_k$ as the result of this phase. Figure 4 shows an example of the hot co-phrase detection process illustrated using $\mathcal{CQ}_k$s only.

3.5 Clustering

The goal of the clustering phase is to create a set of event candidates from the set of hot phrases in the current chunk, $\mathcal{H}_k$ and the set of hot phrase/co-phrase pairs, $\mathcal{O}_k$. Its input consists of $\mathcal{H}_k$, $\mathcal{O}_k$, the tweet records C_k and the timestamp $\hat{\tau}_k$. It returns the set of event candidates in the current chunk $\mathcal{E}_k$.

Essentially, we group phrases related to an event based on their co-occurrence. Specifically, we assert that every hot phrase has a corresponding event candidate:

$$w \in \mathcal{H}_k \Rightarrow \exists! e \in \mathcal{E}_k; w \in S_e \quad (1)$$

and the two phrases of a hot phrase/co-phrase pair are related to the same event candidate:

$$e \in \mathcal{E}_k, w_1 \in S_e, (w_1, w_2) \in \mathcal{O}_k \Rightarrow w_2 \in S_e \quad (2)$$

Initialization: First, we initialize the set of event candidates in the current chunk $\mathcal{E}_k$ to an empty set, create an associative array (map) $\mathcal{L}$ of key-value pairs (y, v) with $y \in$ `str` and $v \in$ `num` which maps each phrase to a numerical cluster label – using the hash function $\hbar$ – and initialize it to an empty map. We also initialize a cluster label counter i to 0. Then, for each hot phrase w in $\mathcal{H}_k$, we add to $\mathcal{L}$ the key w with value i and increment i, thus associating each hot phrase to a distinct cluster (Equation 1).

Agglomerative Clustering: Next, we perform agglomerative clustering: *for every pair of hot phrases $w_1, w_2 \in \mathcal{H}_k$, if there exist hot phrase/co-phrase pairs $(w_1, w') \in \mathcal{O}_k$ and $(w_2, w') \in \mathcal{O}_k$, then we merge the clusters associated with w_1 and w_2 by setting the cluster labels of both hot phrases in $\mathcal{L}$ to the same value* (Equation 2). Particularly, if $\mathcal{L}(w_2) > \mathcal{L}(w_1)$, then we assign $\mathcal{L}(w_2) \leftarrow \mathcal{L}(w_1)$, and vice versa. This prevents oscillation of cluster labels back and forth and ensures termination of the clustering process.

Forming Event Candidates: Next, to form the event candidates, for each cluster label j in $\mathcal{L}$'s set of values, we create a new event representation $e_j = \langle S_j, t_j, D_j \rangle$. Then for all w which $\mathcal{L}$ maps to j, we add w to S_j and for all $(w, w') \in \mathcal{O}_k$, add w' to S_j as well. Thus, the signature of e_j would consist of all the hot phrases labeled as being in cluster j and their hot co-phrases. We set $t_j = \hat{\tau}_k$, and for each tweet record in C_k, if its text (value of the "`TweetText`" field) contains 3 or more phrases of e_j's signature, then we add it to D_j. The choice of value 3 is an empirical heuristic, based on the investigating the data.

We then add e_j to the set of event candidates $\mathcal{E}_k$ and finally return $\mathcal{E}_k$ as the result of this step. Figure 5 shows an example of the clustering process.

3.6 Metadata Extraction and Filtering

In this phase, we extract metadata for event candidates in $\mathcal{E}_k$ and pick interesting events based on the extracted metadata and user-provided editorial rules. The input to this step consists of the set of event candidates for the current chunk $\mathcal{E}_k$, the current chunk's tweet records C_k and the set of user-provided editorial rules. It returns the set of interesting events in the current chunk $\hat{\mathcal{E}}_k$.

Initialization and Metadata Extraction: First, we initialize $\hat{\mathcal{E}}_k$ to an empty set. Then, we go over the tweet records in C_k and for each event candidate e in $\mathcal{E}_k$, we extract the metadata shown in Table 4, which capture various aspects of the candidate event.

For example, to extract $\text{twt}_3(e)$ and $\text{twu}_3(e)$, we initialize $\text{twt}_3(e) = 0$, $\text{twu}_3(e) = 0$ and $\text{TWU}_3(e) =$ which is the set of users posting at least one tweet record in C_k containing 3 or

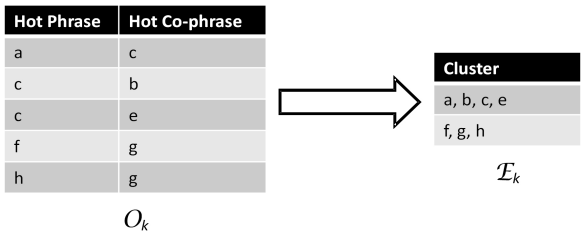

Figure 5: Clustering Example

more unigrams of S_e. Next, for each tweet record tw in C_k, if the tw's "`Tokens`" field contains at least 3 unigrams in S_e, then we increment $\text{twt}_3(e)$ value and add the tw's "`UserID`" field to $\text{TWU}_3(e)$. After going over all tweet records in C_k, $\text{twu}_3(e) = \|\text{TWU}_3(e)\|$.

Application of Editorial Rules: We then apply the editorial rules to to each event candidate's metadata. The editorial rules serve as the means to reflect user's idea about what is interesting and what is not, directly into the results, based on a variety of metadata extracted for each event candidate. Table 5 illustrates the set of editorial rules we use in the current implementation of the system. If an event candidate e passes all the editorial rules, then we pass it to the decision tree classifier.

Classification: The decision tree is trained based on a set of manually-labeled event candidates using the C4.5 algorithm. The training phase starts with drawing a sample of candidate events from a result set of our system. Each sample corresponds to an event candidate and consists of the metadata extracted from the event candidate, along with the threshold μ_u. We then go through the samples one-by-one and check whether it corresponds to a potentially interesting event or not, and labeling it accordingly.

Next, we leverage these samples to train a decision tree, using the C4.5 algorithm implemented in the Weka toolkit[2]. The Weka toolkit outputs source code which we plug into our system programatically and this module could be replaced, for example, in case we want to change the classifier's operation point on the P/R curve.

Using the decision tree, we try to discard systematic noise from the result. Example of the systematic noise in our case includes event candidates which are formed from automatically generated tweets about earning money by filling surveys. If e is classified by the decision tree as being interesting (+), then we add it to $\hat{\mathcal{E}}_k$; otherwise (−) we discard e.

Finally, we return the set of interesting events in the current chunk $\hat{\mathcal{E}}_k$ to the end user. For each event $\hat{e} \in \hat{\mathcal{E}}_k$, we show the following items to the end user:

- S_e as a list of key phrases expressing the event
- t_e as the time when the event has emerged
- a subset of D_e to show the provenance for the event

Figure 6 shows a sample event our system has extracted.

3.7 Remarks

Editorial Rule
$\|S_e\| \geq 2$
$\|S_e\| <= 300000$
$\text{twt}_3 \geq 0$ or $\text{twt}_{all} \geq 0$

Table 5: Editorial Rules Used in The Current System

Algorithm 1 Twitter Event Detection

Input: $\mathcal{C}$ – input tweet chunks, Θ – user-provided parameters
Output: $\hat{\mathcal{E}}$ – extracted events

```
1: Ê ← ∅
2: for k ← h + 1, ··· , m
3:     T_k ← Preprocess and extract uni/bigrams from T_k
4:     H_k ← Extract hot phrases in T_k
5:     O_k ← Extract hot phrase/co-phrase pairs in T_k
6:     E_k ← Cluster O_k and form event candidates
7:     for each e ∈ E_k
8:         Extract metadata for e
9:         if e passes all editorial rules
10:            if e_{k,l} is interesting (decided by decision tree)
11:                Ê_k ← Ê_k ∪ e
12:    Ê ← Ê ∪ Ê_k
13: return Ê
```

Algorithm 1 summarizes our approach to detect emerging events in Twitter. The set of parameters Θ provided by the user are:

- Input tweet files paths
- Stopword list
- Blacklisted phrases
- History length h (in chunks)
- Hot phrase and co-phrase thresholds

$$\mu_f, \mu_u, \lambda_f, \lambda_u,$$
$$\mu'_f, \mu'_u, \lambda'_f, \lambda'_u$$

- Editorial rules

Using these knobs, a user controls which events are emerging, dynamic and interesting. Thus, our system provides a configurable framework to solve the event detection problem in the Twittersphere.

[2]http://www.cs.waikato.ac.nz/ml/weka/

Signature	[default], [boehner], [democrats], [statement], [president], [obama]
Time	Mon Aug 01 2011, 16:10-16:15
Sample of Tweets	New compromise. Obama & Biden resign. Boehner becomes President. Hillary runs in 2012 on a "where are the jobs?" platform.
	Coming up shortly President Obama Statement
	The details will likely disgust liberal dems like myself. RT @AP Obama, Boehner, Democrats strike debt deal to head off government default
	CNN: President Obama will make a statement in the WH briefing Room at approx. 8:40 PM eastern

Figure 6: Sample Extracted Event

Metadata	Description
1gr	Number of unigrams in e's signature
$\text{twt}_p, p = 3, 4, 5$	Number of tweets in C_k containing p or more unigrams of e's signature
twt_{all}	Number of tweets in C_k containing all the unigrams of e's signature
twu_3	Number of users posting tweets in C_k containing 3 or more unigrams of e's signature
twu_{all}	Number of users posting tweets in C_k containing all the unigrams of e's signature
$\text{ret}_{sum}, \text{ret}_{max}, \text{ret}_{min}$	Sum/Max/min of retweets of tweets in C_k containing 3 or more unigrams of e's signature
$\text{rep}_3, \text{reu}_3$	Number of replies and replying users for tweets in C_k containing 3 or more unigrams of e's signature
nws_3	Number of news agencies mentioned in one of the tweets in C_k containing 3 or more unigrams of e's signature
$\text{url}_3, \text{uru}_3$	Number of URLs and number of users posting them in the C_k tweets containing 3 or more unigrams of e's signature

Table 4: Metadata Extracted for Each Event Candidate

4. SCALABILITY

The procedure described in Algorithm 1 could be viewed as a *serial implementation* of our event detection solution: the chunks are processed one after another, by a single thread of execution. As we discussed in the previous section, there are various issues regarding time and memory space usage to be addressed in order to make the system usable for real-volume data:

- Efficient storage of strings - We only store a hash value of (tweet) strings whenever we don't need the actual string. We also retain a single copy of each tweet string and use efficient reference mechanisms to retrieve it whenever needed (through C_k).
- Efficient access paths - Using associative arrays, we try to minimize the delay in accessing various pieces of information. Hashing is our main reference strategy.
- Avoiding quadratic storage costs - As we discussed in Section 3.4, we use the hot phrases to recalculate the phrase/co-phrase pairs' statistics, instead of calculating *all* phrase/co-phrase pairs' statistics in each chunk, to avoid quadratic blow-up of memory usage. Of course, we pay the price of retraversing the tweet records in a chunk multiple times and potentially recalculating some statistics repeatedly.

On the other hand, since one of our primary objectives is to be able to scale up with the input data rate (in the offline mode, to process the data as fast as possible), we want to investigate other implementation strategies to boost the response and processing time. We observe that various parts of Algorithm 1 might be executed in parallel for different chunks since not all of these steps need the results of executing the previous stages of the pipeline on the current or preceding chunks. Next, we express the dependencies among these parts explicitly to leverage parallel execution, followed by the definition of the scheduling problem we aim to solve, to run as many pipeline stages as possible, in parallel.

4.1 Tasks and Dependencies

Let's formalize the interdependencies of different stages of the algorithm by introducing a set of *task types* $\mathcal{T} = \{\text{CL}, \text{PC}, \text{HP}, \text{CC}, \text{HC}, \text{CF}\}$ which are functions corresponding to the various pipeline operations, as follows:

- **Chunk Loading** ($\text{CL}(\Theta) = C_k$) is the task of loading and preprocessing tweets from the disk and returning the next chunk of English tweet records C_k.
- **Phrase Counting** ($\text{PC}(C_k) = \langle \mathcal{PQ}_k, \mathcal{PU}_k \rangle$) is the task of generating phrases of the tweet records in C_k and counting the number of appearences of and the number of users posting each phrase, the result of which is C_k's phrase and user counts $\mathcal{PQ}_k$ and $\mathcal{PU}_k$.
- **Hot Phrase Detection** ($\text{HP}((\langle \mathcal{PQ}_i, \mathcal{PU}_i \rangle)_{i=k-h}^{k}; \Theta) = \mathcal{H}_k$) is the task of determining the hot phrases in C_k based on the statistics of this chunk as well as those of history chunks, calculated in the previous step. It returns a set of hot phrases of C_k, $\mathcal{H}_k$.

- **Co-phrase Counting** ($\mathrm{CC}(\mathcal{H}_k, (C_i)_{i=k-h}^{k}; \Theta) = (\langle \mathcal{CQ}_j, \mathcal{CU}_j \rangle)_{j=k-h}^{k}$) is the task of re-traversing the tweet records in the current chunk and the history chunks to extract the phrase/co-phrase pairs containing a hot phrase in $\mathcal{H}_k$ and recording the number of appearences of and the number of users mentioning each pair, $\mathcal{CQ}_j$s and $\mathcal{CU}_j$s for $j = k-h, \ldots, k$.
- **Hot Phrase/Co-phrase Pair Detection** ($\mathrm{HC}((\langle \mathcal{CQ}_j, \mathcal{CU}_j \rangle)_{j=k-h}^{k}; \Theta) = \mathcal{O}_k$) is the task of determining the hot phrase/co-phrase pairs in C_k, returned as a set $\mathcal{O}_k$.
- **Clustering, Metadata Extraction and Filtering** ($\mathrm{CF}(\mathcal{O}_k, C_k; \Theta) = \hat{\mathcal{E}}_k$) is the task of clustering hot phrase/co-phrase pairs based on common phrases to form candidate events, extracting metadata for each candidate event and filtering the candidate events to obtain the interesting events in C_k, $\hat{\mathcal{E}}_k$.

A *task* is defined by a tuple (η, k) where $\eta \in \mathcal{T}$ and $k \in \mathbb{N}$, to denote running operation η on the chunk C_k. From the definition of tasks, a natural order could be deduced between them which reflects the dependencies of the tasks to each others results. We formalize this intuition as follows.

Definition 4.1 (**Task Partial Order**)**.** For the history length $h \in \mathbb{N}$, we define a partial order among tasks $\leq_h$ as follows:

$$(\eta, k) \leq_h (\eta', k) \Longleftrightarrow \eta \leq \eta'$$
$$(\mathrm{HP}, k) \leq_h (\mathrm{PC}, k-i), i = 1, \ldots, h$$
$$(\mathrm{CC}, k) \leq_h (\mathrm{CL}, k-i), i = 1, \ldots, h$$

for $\eta, \eta' \in \mathcal{T}, k \in \mathbb{N}$ where:

$$\mathrm{CL} \leq \mathrm{PC} \leq \mathrm{HP} \leq \mathrm{CC} \leq \mathrm{HC} \leq \mathrm{CF}.$$

The order defined above is partial since some pairs of tasks (η, i) and (η', j) are not comparable, e.g. $(\mathrm{CC}, 4)$ and $(\mathrm{HP}, 5)$. We will see how this definition is useful when defining a schedule in Section 4.2.

Figure 7 shows a snapshot of the dependencies among tasks for the first five chunks. Each rectangle is one of the functions which is applied to a particular chunk and an arrow from one rectangle to another indicates that executing the latter depends on the result of the former to be available.

Considering the above task dependencies, there are two main strategies to run this algorithm in a parallel fashion:

1. In a *single-machine solution*, we use a multicore machine to run a multi-threaded implementation of the pipeline. We instantiate threads according to an execution plan (or schedule) to make sure that the dependencies are satisfied before running each task and that the waiting time for each task is minimized. This plan should satisfy a set of resource constraints (total available memory and maximum number of concurrently running threads).

2. In a *distributed solution*, multiple machines are used to form a cluster of processing nodes, on which execution agents receive instructions and data from other nodes, run the tasks and distribute the results to peers who need them. The execution plan should consider communication costs as well as data dependencies and waiting time to maximize the speed-up gain.

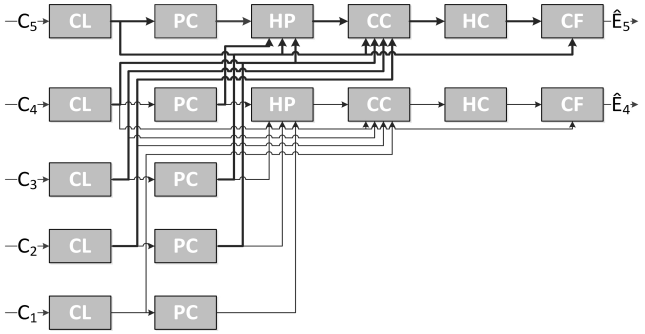

Figure 7: A Snapshot of Dependencies among Tasks. Dependencies for chunk C_5 appear as thicker lines.

In this work, we choose the single-machine approach, mainly because it could serve as a fundamental solution whether we want to use a stand-alone machine or utilize it as a building block (node) in case of a distributed solution.

Our main scalability goals for this single-machine, multi-threaded solution are to run as many tasks in parallel as possible (maximize CPU utility) and optimize the main memory usage to avoid accessing disk as much as possible (maximize memory usage utility). Given the limited resources on a standalone machine, we need to *schedule* the tasks corresponding to the input of the system.

4.2 Scheduling Problem

We define the function $time(\eta, k) : \mathcal{T} \times \mathbb{N} \mapsto \mathbb{N}$ as being the duration of time elapsed running task (η, k). This duration is assumed to correspond to the optimal implementation (in terms of speed and memory usage) of the task (η, k) using a single thread of execution. Now, we define what we mean by a schedule.

Definition 4.2 (**Schedule**)**.** Given a set of input tweet files F, the thread pool of size b indexed by $\mathbb{I}_b$, the definition of tasks and their partial order (see Section 4.1), a schedule is defined as a tuple of functions (ρ_1, ρ_2) where:

$$\rho_1 : \mathcal{T} \times \mathbb{N} \mapsto \mathbb{N},$$
$$\rho_2 : \mathcal{T} \times \mathbb{N} \mapsto \mathbb{I}_b.$$

For each task (η, k) where C_k is extracted from F,

$$(\rho_1(\eta, k), \rho_2(\eta, k)) = (\alpha, \beta)$$

indicates running task (η, k) starting at time $\alpha \in \mathbb{N}$ using the thread instance $\beta \in \mathbb{I}_b$ such that for any pair of distinct tasks $(\eta, k) \neq (\eta', k')$ with $(\rho_1(\eta, k), \rho_2(\eta, k)) = (\alpha, \beta)$ and $(\rho_1(\eta', k'), \rho_2(\eta', k')) = (\alpha', \beta')$:

$$(\eta, k) \leq_h (\eta', k') \Rightarrow \alpha + time(\eta, k) \leq \alpha'$$

and

$$\beta = \beta' \Rightarrow \begin{cases} \alpha + time(\eta, k) \leq \alpha' & \text{if } \alpha \leq \alpha' \\ \alpha' + time(\eta', k') \leq \alpha & \text{otherwise.} \end{cases}$$

So, a schedule determines which task should be run when (ρ_1) and using which thread from the thread pool (ρ_2) .The first constraint ensures that a task is submitted to be run only when all it's dependencies are satisfied and the data it needs to run is ready. The second constraint states that for any pair of distinct tasks which run using the same thread, their run span could not overlap.

The scheduling problem for the single-machine approach could be formalized as follows.

Definition 4.3 (**Single-Machine Scheduling Problem**). Given a set of input tweet files F, the thread pool of size b indexed by $\mathbb{I}_b$, available main memory volume $\mathcal{M}$, the definition of tasks and their partial order (see Section 4.1), find a schedule (ρ_1, ρ_2) to solve the following optimization problem:

$$\underset{(\rho_1,\rho_2)}{\text{minimize}} \sum_{j=1}^{m}(\rho_1(\text{CF}, j) - \rho_1(\text{CL}, j)) + \max_{\eta\in\mathcal{T}, i\in\mathbb{I}_m} \rho_1(\eta, i)$$

such that total amount of memory used at each instant during the execution of the tasks according to the schedule is not larger than $\mathcal{M}$. Here, m is the number of English tweet chunks extracted from F.

A solution to the above problem ensures minimal latency via the first term (summation) and minimal total processing time via the second term of the objective. It is straightforward to verify that in general, a trivial schedule would not solve the above problem.

We have two main choices to solve the above scheduling problem. *Static scheduling* uses mathematical programming techniques (like integer programming methods) to find the best schedule according to the objective and constraints. There are two main issues considering our scheduling problem which makes static scheduling hard to apply:

1. The value of $time(\eta, k)$ function could not be calculated before actually running η on C_k.

2. We wish to be able to port the system along with the scheduling algorithm to online operation mode. In that case as new chunks arrive on-the-fly, we need to re-solve the scheduling problem repeatedly and solving the optimization problem above would be tedious.

Thus, we use a *dynamic scheduling* algorithm, which essentially employs a greedy heuristic to create a schedule on the fly, as new tasks arrive. Next, we describe the dynamic scheduling in more detail.

4.3 Dynamic Scheduling

The goal of the dynamic scheduling is to solve the single-machine scheduling problem (see Definition 4.3) by keeping a priority queue of tasks (see Definition 4.4 below), running the task at the head of this queue whenever the input arguments are ready, chaining the tasks to be executed on each chunk to complete the pipeline, and cleaning up the memory acquired by tasks as soon as possible (i.e. no other tasks need the results afterwards). The input of the dynamic scheduling module consists of the paths to the input tweet files, the set of user-provided parameters Θ and the size of the thread pool b (the maximum number of concurrent threads of execution).

In order to be able to order *all the tasks* to be executed upon F, we need a *total order* using which we are able to compare *every pair of tasks*. Hence, we define another order among tasks as follows.

Definition 4.4 (**Task Total Order**). We define a total order among tasks $\leq$ as follows:

$$(\eta, i) \leq (\eta', j) \iff i < j \vee ((i = j) \wedge (\eta \leq \eta'))$$

for $\eta, \eta' \in \mathcal{T}, i, j \in \mathbb{N}$ where:

$$\text{CL} \leq \text{PC} \leq \text{HP} \leq \text{CC} \leq \text{HC} \leq \text{CF}.$$

Initialization: First, we instantiate the following queues and sets:

- A priority queue **WQ** which contains the tasks waiting to be run. This is the main queue from which the scheduler picks the next task to run. It is initialized to an empty queue.

- A set **RS** containing the tasks the execution of which is in progress. This set is used to check whether the preconditions of the next task to run are satisfied; i.e. the inputs to the next task are ready. It is initialized to an empty set.

- A set **HS** which contains the tasks that have high priority, but their input arguments are not computed yet. The contents of this set would be enqueued to **WQ** after the next task to run is picked. It is initialized to an empty set.

Also, we create a *thread pool* of size b to run the tasks.

Running The Tasks: Next, we insert the chunk loading task $(\text{CL}, 1)$ into **WQ**, to load the first chunk C_1 into memory. Then, while there is a task to be run, the following two steps are executed in a busy loop. During step 1, while $\|\mathbf{WQ}\| > 0$ and $\|\mathbf{RS}\| < b$, we dequeue the head of **WQ** and check whether its preconditions are satisfied (i.e. its arguments are already calculated). If the inputs are ready, then we submit the task to the thread pool to be executed, add it to **RS**, move all elements of **HS** back to **WQ** and move to step 2. On the other hand, if the inputs are not ready, then the task is added to **HS** and the next task is polled from **WQ**, repeating step 1.

Chaining and Cleaning Up: In step 2, we go over the running tasks in **RS** and check whether any of them is already finished. If there is such a task (η, i), then we remove it from **HS**, clean the memory space occupied by this task's data items which is no longer needed (no currently-running or future tasks depend on them), and insert the next task in the pipeline, to be applied to chunk C_i, into **WQ** (chaining). The only exception is for tasks (CL, i) upon completion of which we insert two new tasks into **WQ**, namely $(\text{CL}, i+1)$ to load the next chunk and (PC, i) to continue executing the pipeline on the current chunk.

5. EMPIRICAL EVALUATION

We now present the experimental results obtained by running the system upon real-volume Twitter data, in order to evaluate event detection accuracy and timing, the contribution of each major component of the system and analyze the sensitivity of the system to changes in parameter values.

Dataset: We have obtained two datasets of all tweets posted during a particular period of time and received from the Twitter firehose. The first dataset consists of all the tweets

posted during July 31, 2011 in form of 24 zip files (corresponding to the the 24 hours of the day) with the total volume of about 66 GB, obtained from Kosmix (which was later acquired by Walmart®[3]). Each line of these files (after unzipping) corresponds to a tweet and is string representation of a JSON object. It consists of various fields obtained from the Twitter firehose as well as extra fields added by the Kosmix softwares.

The second dataset consists of all tweets posted during the period between Aug 1, 2012 and Aug 14, 2012 inclusive. Each day worth of tweet is stored in a zip file, accounting for the total of 14 zip files of the total volume around 344 GB, obtained from the @WalmartLabs®[4]. Each line of these files (after unzipping) corresponds to a tweet and is a tab-separated string of various data fields representing tweet information. It consists of various fields obtained from the Twitter firehose as well as extra fields added by the @WalmartLabs softwares.

Golden Data: For our experiments, we have sampled four 2-hour long periods worth of tweet as follows:

- July 31, 2011 from 16-18PM
- August 3, 2012 from 16-18PM
- August 6, 2012 from 16-18PM
- August 8, 2012 from 16-18PM

For each of these periods, we extract all the tweets from the corresponding zip files and parse each tweet using customized parsers (see Section 5.2). Then, we use the timestamp of the tweet (which shows the time it has been posted) to decide whether it should be included in the sample or not. The tweets in each of these periods are then classified using the English langage classifier (described in Section 3.2) and every consecutive sequence of 300K English tweets is written to a zip file on disk as a chunk. Table 6 shows various metrics of each periods chunks.

To measure the accuracy of the system, we need to come up with the set of interesting events that have been mentioned on the Twitter during the periods of interest (called *golden events* hereafter). To do this, we have run our system with various parameter sets, using a minimal editorial rule set (Table 5) and no decision tree classification. We then go through these results manually and investigate each for being an interesting event or not, by looking up news articles, reading the corresponding tweets, etc. The use of minimal editorial rule set prevents the result set from getting too large and hard to investigate manually.

Table 8 summarizes the statistics of this step for various parameter sets shown in Table 7. We emphasize that the golden event set (extracted interesting events) does not include any redundancies: if a particular interesting event is detected in more than one chunk, we only add the first one to the golden event set.

Training the Decision Tree: In order to prepare the training data for the decision tree, we first sample 10 different values for $\mu_f = \mu_u$ uniformly in the $[50, 700]$. Then, we multiply each value by 0.65 and take the ceiling of the multiplication results to compute $\mu'_f = \mu'_u$ values. The values of λ_f, λ_u, λ'_f and λ'_u are set to the same values as shown in Table 7.

Next, we run the system on the sampled data. using these thresholds and the editorial rules illustrated in Table 5, and sample 50 candidate events from the output of each run along with the metadata extracted from the corresponding chunks. For each sample, we use the metadata as well as the value of $\mu_f = \mu_u$ as the features corresponding to the sample and add them all to the training set, which in turn contains 500 training samples. We then manually label these samples, which yields in 261 samples being interesting (+) and 239 samples not being interesting (−).

We then train a decision tree on these samples using the C4.5 algorithm (implemented in the Weka toolkit) by 10-fold cross-validation. The precision of the trained decision tree is 0.85(209/244), the recall is 0.80(209/261) and the F1-score is 0.82. This decision tree is then incorporated into the pipeline to finish the end-to-end system.

Methodology: We feed the whole labeled (sampled) dataset into the end-to-end pipeline using the parameter sets of Table 7 and collect the results. The results are then investigated manually to measure the precision and recall of the system. As mentioned before, in this study we have stored the chunks separately on disk, so the timing results do not include the language classification time. Since we write the actual tweets to the chunk files, we still need to parse the tweets to feed them to the pipeline, so the parsing time is included.

To measure the response time of the system, we repeat the above process 10 times on the sampled dataset for each parameter set. This also helps us assess the sensitivity of the system to variations in the parameters. In this study, we have manually set the thread pool size b for each parameter set as the maximum number of threads that could be run without triggering the use of virtual memory. We see an automated way of adjusting b to the available memory, number of CPU cores and current status of the system, as an interesting study for future work.

5.1 Performance Measures

Accuracy: We summarize the accuracy measures of the system for various parameter sets in Table 9. To calculate precision, we have divided the number of candidate events which refer to a golden event *in the correct chunk*, by the total number of candidate events produced by the complete pipeline. This means that if a golden event is extracted from the chunk C_k for the first time, but the first time an output candidate event referring to it is detected in a chunk C_{k+i} and $i > 0$, then the output candidate event is counted as a false positive.

Calculating recall, we have divided the the number of golden events corresponding to which a candidate event is detected in the correct chunk, by the total number of golden events. Again, this means that if the first candidate event referring to a particular golden event is detected in a later chunk than that of the golden event, we mark the golden event as undetected (false negative).

Total Processing Time: We report total processing time of the whole pipeline on a commodity machine (see Section 5.2 for the actual configurations) in Figure 8.

[3]http://www.walmart.com/
[4]http://www.walmartlabs.com/

Time Period	Number of Chunks (ea. 300K tweets)	Total Data Volume on Disk (GB)
July 31, 2011, 16-18PM	23	2.3
August 3, 2012, 16-18PM	31	1.4
August 6, 2012, 16-18PM	30	1.3
August 8, 2012, 16-18PM	31	1.4

Table 6: Chunks Properties

	μ_f	μ_u	λ_f	λ_u	μ'_f	μ'_u	λ'_f	λ'_u
Parameter Set 1	300	300	1.3	1.3	250	250	1.1	1.1
Parameter Set 2	500	500	1.3	1.3	300	300	1.1	1.1

Table 7: Parameter Sets (Hot Phrase and Hot Co-Phrase Threshold Values)

Latency:

5.2 Technical Details

Parsers: For the first dataset, we observe that using a general-purpose JSON parser has a considerable overhead, since it reconstructs the whole JSON object. Thus, we have written a fairly simple special-purpose JSON parser which only extracts the fields we need for our analysis (see Section 3.2 and Table 2). Using this parser, we have reduced the parsing time considerably comparing to the general-purpose JSON parsers.

Parsing the second dataset is semantically straightforward, but the main problem with the tweets in these files is that a lot of tweets contain values which include one or more tab characters (the delimiter). This makes it nontrivial for the parser to extract the correct values of the fields since, unlike the JSON format, the boundaries of the fields are no longer distinguishable. We have leveraged information about particular fields (value type and format) in our tab-separated parser to reestablish the field boundaries when possible and detect the cases where the discrepancy is too much that we need to drop the tweet altogether.

Hardware Specification: The program is run on a machine with 24GB of RAM, equipped with a 12-core, dual socket processor. We use a fixed thread pool to execute tasks to reuse the thread instances and avoid the thread creation overhead.

6. RELATED WORK

Research work about event detection in Twitter might fall into two broad categories of detecting *general* and *class-specific* events. General event detection looks for *any* conversation which satisfies particular event properties, e.g. bursty or hot, whereas class-specific frameworks only consider particular categories of events, like social events, controversial events and disasters.

Class-specific event detection systems aim to address specific problems like creating calendars, effectively managing disasters and predicting or reacting to social and political controversies. Some of these approaches look for events in individual tweets. Sakaki et al. [17] aim to detect whether an earthquake is happening in real-time. To do so, they periodically query Twitter with a set of prespecified keywords (using the search API) to obtain tweets potentially related to earthquake updates. From each tweet, they extract various statistical, textual and contextual features and use those features to classify the tweet using a support vector machine for being related to an ongoing earthquake or not. For earthquake-related tweets, they try to estimate the location of the earthquake by applying particle filters to GPS information accompanying the tweets or the registered locations of tweeting users. The result would be earthquake alerts which are sent out to registered users. Ritter et al. [16] try to extract structured representations of events which are highly correlated with a particular date. They POS-tag each tweet and then try to extract triplets of the form $\langle$*named entity*, *event phrase*, *date*$\rangle$ from it. The extracted triplets are considered as event candidates and a variant of latent Dirichlet allocation (LinkLDA) topic model is used to determine the event type. They also rank event candidates according to a measure of significance and the top-rank, typed events are presented to the end user in a calendar format. Other similar approaches include [4, 10, 19]

Another group of class-specific approaches work with collections of tweets (tweet clusters.) Petrovic et al. [13] try to detect new stories discussed in a stream of messages in a real-time fashion. For any incoming tweet, they use a space/time-bound version of locality sensitive hashing to find the closest tweet seen so far (in a bounded period of time). If the closest tweets is farther than a particular threshold, then the new tweet is detected as a *first story* tweet which initiates a new thread of discussion. The user gets notified about first stories as they are detected. Popescu et al. [15] tackle the problem of identifying controversial events, which are defined as the events provoking opposing public discussions. They start with a set of entities (for example, celebrities) and collect the tweets referring to any of them from the Twitter firehose, for a particular time period (day). For each entity, they form a *snapshot*, which is a triplet of form $\langle$*entity*, *day*, *set of tweets*$\rangle$. They then use a gradient-boosted decision tree, trained on a wide range of linguistic, structural and social features of the snapshots, to assign a controversy score to each snapshot. The top-ranked snapshots are presented to the end user as controversial events. Phuvipadawat et al. [14] aim to identify breaking news from the Twitter messages. They query the Twitter API with a predefined set of keywords (like "#breakingnews"and "breaking news") and build a content-based index on the obtained tweets. Messages with similar content (based on TF-IDF representation) are grouped as breaking

	Total Number of Extracted Candidate Events	Total Number of Extracted Interesting Events
Parameter Set 1	145	51
Parameter Set 2	99	36

Table 8: Golden Data Statistics

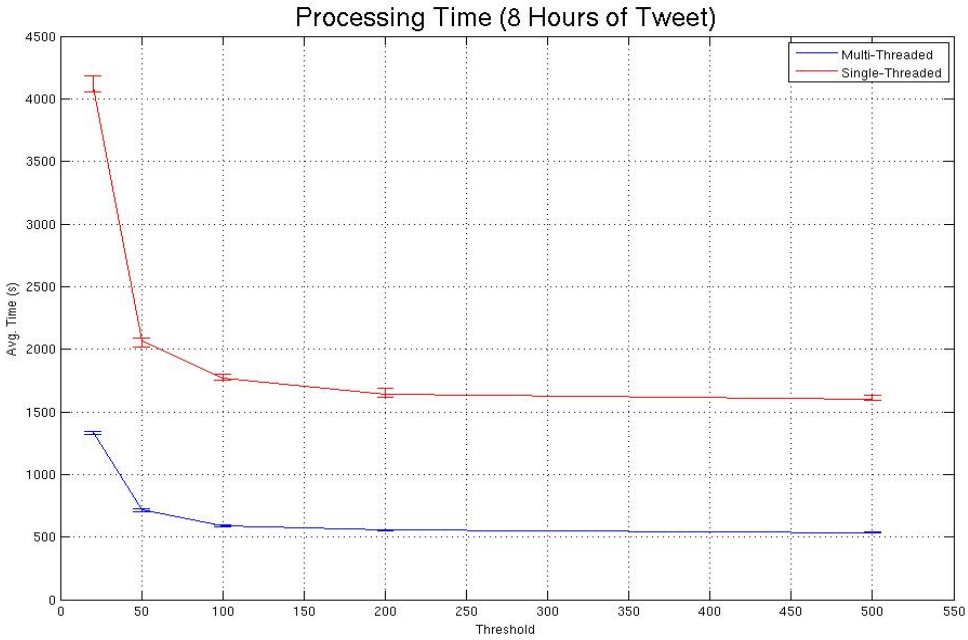

Figure 8: Timing Results

stories and the groups are score with respect to number of retweets and number of users posting about the story. The stories are then enriched by external sources for news URLs and media items and the top-rank stories are returned to the user. Other examples of this type of systems include [18, 6].

General event detection systems use a wide range of ideas and algorithms to extract hot emerging events from the Twitter stream. Most of these approaches rely on the notion of *burstiness*. Systems like [20, 9, 1, 5] follow a graph-theoretic approach to detect events. Weng et al. [20] try to detect events by tracking and grouping bursty keywords in the Twitter stream. They generate a *burst signal* for each unigram appearing in a chunk of tweets and discard the ones with almost flat signals (low entropy.) The signals are then used to create a modularity matrix based on cross-correlation similarity measure. The graph, represented by the modularity matrix, is subsequently partitioned using spectral clustering into subgraphs which are in the form of keyword sets (event candidates.) These candidates are scored and ranked to be returned to the user. Li et al. [9] try to overcome the limitations of [20] using simpler, less computation-intensive algorithms (no wavelet analysis) and more thorough content analysis (processing segments instead of unigrams), to improve scalability and detection accuracy. They extract, from each window of the Twitter stream, the bursty segments (of tweets) along with a bursty probability (a measure of how bursty the segment is) and the number of users who have mentioned the segment in their tweets. They form a segment similarity matrix by calculating the pairwise similarities among the tweets containing these segments and apply graph clustering to this similarity matrix to generate segment clusters as candidate events. A last filtering stage retains significantly important event candidates with respect to a *newsworthiness* measure, which are returned to the user.

Agarwal et al. [1] address the problem of real-time emerging event detection leveraging a relaxed version of graph clustering on correlated keyword graph. They build a node-weighted, edge-weighted active keyword graph from the set of bursty keywords in the current window, each node of which represents a bursty keyword weighted by the number of users posting about it and each edge of which is weighted based on the Jaccard distance between the set of users tweeting about the keywords. This graph is in turn partitioned into dense clusters to form keyword groups as events, which are ranked accordingly and reported to the end user. Cataldi et al. [5] tackle the problem of emerging topic detection in real-time. They propose an aging theory-based approach to monitor keywords over consecutive time intervals and identify emerging keywords according to their frequency and authority of the users posting about them. Leveraging the co-occurrence of the keywords in tweets, they build a keyword-based *topic graph*, the strongly connected

	Precision	Recall	F1 Score	Number of Candidate Events Discarded by The Decision Tree
Parameter Set 1	0.94	1	0.97	21
Parameter Set 2	0.98	1	0.99	15

Table 9: Accuracy Results

components of which comprise emerging topics during each particular period. The emerging topics are ranked and returned to the user as keyword sets.

Besides the graph-theoretic approaches, other techniques are utilized for general event detection in Twitter as well. Becker et al. [3] address the problem of online identification of real-world events and the associated content posted on Twitter. They process, on an hourly basis, the incoming tweet feed by using an incremental clustering algorithm to create message clusters as event candidates. They then extract various features of different categories (temporal, social, topical and Twitter-centric) from each cluster which are used to classify the event candidates as events or non-events accordingly. Select top events are displayed to the end user along with snippets of the associated tweets. Mathioudakis et al. [12] try to detect emerging topics from the Twitter stream along with meaningful descriptions for each topic. They use queuing theory to find bursty keywords which are then greedily clustered, based on co-occurrence in the recent tweets, to form *trends*. These trends are enriched with descriptions (using PCA or SVD), news source citations and geographical origins of tweets to be shown to the end user. Alvanaki et al. [2] tackle the problem of emergent event detection by identifying shifts in the correlation between tags. They use popular tags and named entities (extracted from the tweets) to generate tag pairs (containing at least one popular tag) or hot-tag/entity pairs as candidate topics and then track the correlation of each tag pair over time to detect significant increase in their co-appearance. The hot tag pairs/tag-entity pairs are scored based on the shift strength and the top-k pairs are presented to the user as emergent topics. Other general event detection systems include [7, 11, 8].

As could be observed from the above discussion, one of the popular architectures to detect events is based on a two-phase strategy as follows. In the first phase, a set of *emerging topics/contexts* is built, usually by clustering tweets or segments/terms (called in [20] *document-pivot* and *feature-pivot* approaches respectively.) These topics are naturally accompanied by a provenance of tweets which might be organized in various ways. The purpose of the second phase is to distinguish between topics that correspond to particular events from those that are non-event-related discussions. A common approach would be to extract various features from clusters and their provenance and use a classification technique to label the topics as event-related and non-event-related.

7. REFERENCES

[1] 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*, 5(10):980–991, 2012.

[2] F. Alvanaki, S. Michel, K. Ramamritham, and G. Weikum. Enblogue: emergent topic detection in web 2.0 streams. In T. K. Sellis, R. J. Miller, A. Kementsietsidis, and Y. Velegrakis, editors, *SIGMOD Conference*, pages 1271–1274. ACM, 2011.

[3] H. Becker, M. Naaman, and L. Gravano. Beyond trending topics: Real-world event identification on twitter. In *Proceedings of the Fifth International Conference on Weblogs and Social Media, Barcelona, Catalonia, Spain, July 17-21, 2011*. The AAAI Press, 2011.

[4] E. Benson, A. Haghighi, and R. Barzilay. Event discovery in social media feeds. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies - Volume 1*, HLT '11, pages 389–398, Stroudsburg, PA, USA, 2011. Association for Computational Linguistics.

[5] M. Cataldi, L. Di Caro, and C. Schifanella. Emerging topic detection on twitter based on temporal and social terms evaluation. In *Proceedings of the Tenth International Workshop on Multimedia Data Mining*, MDMKDD '10, pages 4:1–4:10, New York, NY, USA, 2010. ACM.

[6] A. Cui, M. Zhang, Y. Liu, S. Ma, and K. Zhang. Discover breaking events with popular hashtags in twitter. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, CIKM '12, pages 1794–1798, New York, NY, USA, 2012. ACM.

[7] E. Ilina, C. Hauff, I. Celik, F. Abel, and G.-J. Houben. Social event detection on twitter. In *Proceedings of the 12th international conference on Web Engineering*, ICWE'12, pages 169–176, Berlin, Heidelberg, 2012. Springer-Verlag.

[8] S. P. Kasiviswanathan, P. Melville, A. Banerjee, and V. Sindhwani. Emerging topic detection using dictionary learning. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, CIKM '11, pages 745–754, New York, NY, USA, 2011. ACM.

[9] C. Li, A. Sun, and A. Datta. Twevent: segment-based event detection from tweets. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, CIKM '12, pages 155–164, New York, NY, USA, 2012. ACM.

[10] R. Li, K. H. Lei, R. Khadiwala, and K. C.-C. Chang. Tedas: A twitter-based event detection and analysis system. In A. Kementsietsidis and M. A. V. Salles, editors, *ICDE*, pages 1273–1276. IEEE Computer Society, 2012.

[11] H. Ma, B. Wang, and N. Li. A novel online event analysis framework for micro-blog based on incremental topic modeling. In *Software Engineering, Artificial Intelligence, Networking and Parallel Distributed Computing (SNPD), 2012 13th ACIS*

International Conference on, pages 73–76, 2012.

[12] M. Mathioudakis and N. Koudas. Twittermonitor: trend detection over the twitter stream. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 1155–1158, New York, NY, USA, 2010. ACM.

[13] S. Petrović, M. Osborne, and V. Lavrenko. Streaming first story detection with application to twitter. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, HLT '10, pages 181–189, Stroudsburg, PA, USA, 2010. Association for Computational Linguistics.

[14] S. Phuvipadawat and T. Murata. Breaking news detection and tracking in twitter. In *Web Intelligence and Intelligent Agent Technology (WI-IAT), 2010 IEEE/WIC/ACM International Conference on*, volume 3, pages 120–123, 2010.

[15] A.-M. Popescu and M. Pennacchiotti. Detecting controversial events from twitter. In *Proceedings of the 19th ACM international conference on Information and knowledge management*, CIKM '10, pages 1873–1876, New York, NY, USA, 2010. ACM.

[16] A. Ritter, Mausam, O. Etzioni, and S. Clark. Open domain event extraction from twitter. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '12, pages 1104–1112, New York, NY, USA, 2012. ACM.

[17] T. Sakaki, M. Okazaki, and Y. Matsuo. Earthquake shakes twitter users: real-time event detection by social sensors. In M. Rappa, P. Jones, J. Freire, and S. Chakrabarti, editors, *WWW*, pages 851–860. ACM, 2010.

[18] B. Tsolmon, A.-R. Kwon, and K.-S. Lee. Extracting social events based on timeline and sentiment analysis in twitter corpus. In G. Bouma, A. Ittoo, E. Mtais, and H. Wortmann, editors, *NLDB*, volume 7337 of *Lecture Notes in Computer Science*, pages 265–270. Springer, 2012.

[19] K. Watanabe, M. Ochi, M. Okabe, and R. Onai. Jasmine: a real-time local-event detection system based on geolocation information propagated to microblogs. In C. Macdonald, I. Ounis, and I. Ruthven, editors, *CIKM*, pages 2541–2544. ACM, 2011.

[20] J. Weng and B.-S. Lee. Event detection in twitter. In *Proceedings of the Fifth International Conference on Weblogs and Social Media, Barcelona, Catalonia, Spain, July 17-21, 2011*. The AAAI Press, 2011.