Fast, Scalable, and Context-Sensitive Detection of Trending Topics in Microblog Post Streams

NARGIS PERVIN, FANG FANG, ANINDYA DATTA, and KAUSHIK DUTTA,
National University of Singapore
DEBRA VANDERMEER, Florida International University

Social networks, such as Twitter, can quickly and broadly disseminate news and memes across both real-world events and cultural trends. Such networks are often the best sources of up-to-the-minute information, and are therefore of considerable commercial and consumer interest. The trending topics that appear first on these networks represent an answer to the age-old query "what are people talking about?" Given the incredible volume of posts (on the order of 45,000 or more per minute), and the vast number of stories about which users are posting at any given time, it is a formidable problem to extract trending stories in real time. In this article, we describe a method and implementation for extracting trending topics from a high-velocity real-time stream of microblog posts. We describe our approach and implementation, and a set of experimental results that show that our system can accurately find "hot" stories from high-rate Twitter-scale text streams.

Categories and Subject Descriptors: H.3.4 [**Information Storage and Retrieval**]: Systems and Software

General Terms: Algorithms, Design, Performance

Additional Key Words and Phrases: Trending topics, microblogs, Twitter, scalability

ACM Reference Format:
Pervin, N., Fang, F., Datta, A., Dutta, K., and Vandermeer, D. 2013. Fast, scalable, and context-sensitive detection of trending topics in microblog post streams. *ACM Trans. Manage. Inf. Syst.* 3, 4, Article 19 (January 2013), 24 pages.
DOI: http://dx.doi.org/10.1145/2407740.2407743

1. INTRODUCTION

Real-time social networks such as Twitter have become a powerful means of disseminating emerging news [Kwak et al. 2010], often well before traditional media can confirm and report on the events. When a post of widespread interest arrives on Twitter, recipients react in any number of ways, including reposting the original post or creating a new post with added commentary. As the news travels across the "Twittersphere," more and more people post on the same topic, making it *trending* [Johnson 2009; Popescu and Pennacchiotti 2011]. The greater the interest in the topic, the stronger the trending topic will be in the Twitter stream. These trending topics represent an answer to the age-old query "what are people talking about?", and have been well discussed in the literature [Cataldi et al. 2010; Hotho et al. 2006; Mathioudakis and Koudas 2010].

Given the level of interest in "what's new," many content publishers have trending topic segments. Recent impactful examples include The Washington Post, CNN.com, and The Los Angeles Times. However, they are, (a) limited to the coverage of that particular publication, and (b) not timely—topics show up long after they have become

Author's address: K. Dutta; email: duttak@nus.edu.sg.

DOI: http://dx.doi.org/10.1145/2407740.2407743

Table I. Twitter Trending Topics vs TrendMiner Trending Topics

Trending Topics Reported by Twitter	Trending Topics Reported by TrendMiner
Harry Redknapp	[Harry, Redknapp, cleared, tax, evasion]
#syria	[russia, china, veto, u.n., security, council, resolution, syria]
Ahmadinejad	[iran, parliament, summons, president, Ahmadinejad, economy, policy]
super bowl	[giants, win, super, bowl]
#Aleppo	[suicide, bombings, aleppo, syria, wound, 200, kill, 28]
#syria	[syria, rejects, arab, peace, mission, proposal]

popular. For example, we consider the recent *CNN News pulse* feature, which attempts to provide trending topics. While this feature does present trending topics, it is still only of limited scope, in that it covers only topics drawn from stories appearing on CNN.com, which publishes approximately 10–15 new stories per hour [Jay Sandhaus Lead Architect, CNN 2010 personal conversation].

Twitter, as a microblogging service, also attempts to provide trending topics. This feature considers all posts to the service, a massive-scale problem indeed, but provides only very high-level topics. Consider, for example, the first column of Table I, which presents a set of trending topics identified by Twitter's trending topics feature in January, 2012. While these trending topics do identify important and timely subjects, they are consistent in their lack of context—there is virtually no sense of what is important about the trending topics. Consider, for example, the "super bowl" trending topic identified by Twitter. Clearly, the championship game is important, but it would be much more helpful to know which team won the game.

In this article, we propose *TrendMiner*, a new and context-sensitive method of detecting trending topics in a microblog post stream. Our system is designed to identify trending topics with a higher level of contextual meaning than current trending topic-detection methods. Let us compare the the first column of Table I, which shows Twitter-generated trends, to the second column, which shows a corresponding set of trending topics detected by TrendMiner. Clearly, TrendMiner's detected trending topics provide higher context, and overall a better sense of what is currently important and why it is important—for example, TrendMiner was able to identify the winner of the Super Bowl.

What is it about the phrases in the second column that characterizes greater context than those in the first in the first column? It is well known that noun-verb phrases provide greater context than just noun or noun phrases alone [Khader et al. 2003] (like those topics provided by Twitter's trend feature, which contains only noun phrases). The identification of such meaningful high-context word clusters to describe trending topics is our focus in this work; specifically, in this article, we propose a system for detecting trending topics from Twitter posts with greater context than existing trend-detection systems.

At this point, it is worth considering the question of why current solutions provide so little context. In fact, the scale and timeliness constraints imposed by the Twitter scenario represent key impediments to workable practical solutions to detecting trending topics with higher context. Of these constraints, the scale of the Twitter trending topic detection problem is paramount. The volume of streaming Twitter posts to process is massive—Twitter post activity currently generates on average 340 million posts per day (or one billion posts every three days) [Twitter, Inc. 2012]. The volume of posts can spike to more than double this rate during special events and breaking news stories. For instance, during the August, 2011, MTV Video Music Awards, the post rate to Twitter rose to 8,868 posts per second (when adjusted to a daily post rate, this works out to more than 766 million posts per day) [Twitter, Inc. 2011]. While simply analyzing the incoming stream of posts represents a formidable

challenge, the Twitter scenario also imposes a significant time constraint—it is important to be able to detect a trending topic while it is still trending.

In the context of this massive scale problem, it is worth considering user expectations for online responsiveness. Recent reports [Forrester 2012; Rui and Whinston 2012] note that online systems need to provide responses within 2 seconds to meet user expectations for application responsiveness.

Our proposed system provides such online responsiveness for user queries for trending topics. Under a variety of operating conditions (described in detail in Section 7), our system is consistently proven to be fast (i.e., sub-2-second responses times for user queries) and scalable, capable of identifying trending topics within 2–3 seconds of the associated posts arriving at the system. Most importantly, our approach is demonstrated to be substantially (150%) more efficient than the current state of the art, while still providing strong accuracy (high recall overall, and high precision when the number of words in the trending topic is relatively small). In summary, our implementation represents one of the first systems capable of scalable, high-context trending topic detection from microblogging social networks.

The rest of this article is organized as follows. Section 2 places our work in the context of related work. Sections 3 and 4 describe the intuition and details of our approach, respectively. Section 5 describes an analysis of the time and space complexity of our approach, while Section 6 discusses an analysis of the expected scalability of our system. Section 7 presents a set of experimental results describing the performance and scalability of our implementation. Section 8 concludes the article.

2. RELATED WORK

Work related to our focus in this article falls into two categories. We first consider the broad area of mining data streams, and then consider the specific area of detecting trending topics in microblogging data streams.

Broadly, mining frequent word clusters from a text stream can be viewed as a variant of mining frequent itemsets from a data stream—posts can be viewed as transactions, and words within posts can be viewed as items. Based on the stream data processing model [Zhu and Shasha 2002], itemset-mining approaches fall into three categories [Li and Lee 2009]: *landmark-window based mining*, *damped-window based mining*, [Giannella et al. 2004] and *sliding-window based mining*. In the landmark-window model [Manku and Motwani 2002], mining methods consider an interval between a landmark and the current time. In the damped-window model [Chang and Lee 2003; Giannella et al. 2004], windows are assigned weights, such that newer data is weighted more heavily than older data. Methods in the sliding-window model [Chang and Lee 2004; Chi et al. 2004; Li et al. 2009] consider a fixed-size time window of the most recently streamed data. Our work follows the sliding-window approach in general, where the window size for clustering is specified by the user's desired query timeframe.

At a high level, our strategy is similar to the Apriori approach [Agrawal and Srikant 1994], which was designed to derive association rules from a dataset. The similarity is limited to the method used for generating potential candidates; our method differs significantly from the Apriori approach in pruning the potential cluster set—in filtering out infrequent clusters. In the Apriori approach, the database must be scanned to count the newly generated clusters. It is impossible to apply this in the Twitter scenario, due to the high rate of arrival of incoming posts that require processing—it would require significant time to simply count the frequency of clusters. Further, the volume of posts is so massive that we do not store the posts received after initial processing, making it impossible to count the frequency of sets of terms at a

later time. In order to handle this, we store only the frequencies of two-word clusters. Based on these frequencies, we approximate the frequencies of larger clusters. This approximation increases the efficiency dramatically, while still retaining high accuracy.

We now move on to consider work specifically in the area of detecting trending topics in microblogging data streams. Such work may be found in three recent papers [Benhardus 2010; Cataldi et al. 2010; Mathioudakis and Koudas 2010]—all three propose methods for identifying trending topics on Twitter. We consider each in turn, and then consider a few more distantly related works.

Benhardus [2010] proposes a method to determine the trending topics using an approach based on TF-IDF scores. This method considers only unigram and bigram word clusters as potential trending topics, and therefore is limited in scope. In contrast, our approach, which can determine topics containing more than two grams, provides more flexibility in the result set. Further, our approach provides online trending topic querying capabilities, which Benhardus' approach does not.

Mathioudakis and Koudas [2010] present TwitterMonitor, a system to detect trending topics in Twitter. This system detects bursty keywords and then groups these keywords to form clusters. While we would like to compare our work with this short paper, it lacks detail, both algorithmic as well as experimental. However, what is made clear in this paper is that a single pass over the Twitter stream is not enough—to extract bursty keywords, one pass is made over incoming posts to identify bursty singleton words and then, for each identified bursty keyword, a second pass is made over recent posting history to group keywords together. Given that there may be many bursty words even in small time periods, this could prove to be extremely computationally expensive.

Cataldi et al. [2010] have developed metrics to individually identify each word that might indicate a trending topic. Their method applies these metrics, then groups the words by computing correlations across them. This correlation computation is highly compute-intensive; performing such computations for each pair of potentially trending words requires significant effort. Although this approach may generate results of reasonable quality, it is not suitable for scenarios requiring real-time performance, which is our focus in this research. Instead of utilizing exact statistical correlation, as is the case in Cataldi et al. [2010], we rely on heuristics to develop an approach that can operate online in near real time. In Section 7, we describe a set of experimental results that demonstrate the scalability differences between the two approaches—the differences are substantial.

Asur and Huberman [2010] took a slightly different approach to trending topic detection using a stochastic model to find the growth of trending topics. They collected the topics from Twitter using the search API and examined the factors associated with trending topics. They showed that the distribution of trending topics is lognormal, and found that the trending topics are primarily posts that are reposted frequently by others. This work is complementary to ours, and represents a potential avenue for improvement in our work.

Kannan et al. [2010] propose the TrendTracker system, which implements real-time visualization to display emerging trending topics collected from Twitter. Varying window sizes indicate trendiness, where trendier topics are shown in larger windows, allowing users to check current trending topics easily. This work is primarily focused on displaying the output of a trending topic detection method, and thus is also complementary to our work here.

Other works in microblog analysis consider predicting movie revenues using social media [Asur and Huberman 2010; Rui and Whinston 2012], and sentiment analysis of Twitter posts, to model emotive trends and economic indicators [Bollen et al. 2011].

3. INTUITION AND OVERVIEW

We are interested in identifying and quantifying trending topics in popular microblogging social networks, such as Twitter, with online responsiveness. A trending topic constitutes a set of words or word phrases that is experiencing an increase in usage with respect to its long-term usage [Benhardus 2010]. Following the lead of recent research with similar goals [Cataldi et al. 2010; Mathioudakis and Koudas 2010], we model a topic as a *cluster of keywords*, and the notion of trending via the concept of *burstiness* [Glance et al. 2004; Liang et al. 2010]. As a result, identifying trending topics reduces to a two-step process: (1) finding word clusters that capture topics discussed in the stream of incoming posts; and (2) identifying *bursty clusters*, i.e., those that occur more frequently than usual within a particular time period.

Our approach is more generic than currently proposed methods, in that we support the ability to perform general trending topics queries, given any user-specified time interval. In other words, our method can support the identification and display of current hot topics, but it can also identify the trending topics from an arbitrary time period in the past. We model Step (2) the identification and display of hot topics, as a general query of the following form: *retrieve trending topics in a time interval [begin_time, end_time]*. Identifying current trending topics is a special case of this query with *end_time* set to the current time. This introduces significant additional time and space complexity with respect to the extant solutions: (a) substantial additional processing is required to answer the historical queries, and, (b) historical data will need to be stored in secondary storage, making data access expensive. Under these constraints, a straightforward implementation of this two-step process would not be scalable or responsive enough.

The crux of our approach is based on the observation that the rate of Twitter posts arriving at the system is likely to be several orders of magnitude larger than the rate at which queries arrive at the system. Thus, the key to online scaling of the system is to reduce the stream-processing work as much as possible. This insight is the foundation of the *TrendMiner* approach: we have implemented a lazy evaluation strategy in which the incoming stream is minimally processed, and much of the cluster computation occurs in conjunction with burstiness identification when a query to identify a trending topic is processed. This is very different from existing solutions, which attempt to create the clusters as posts stream in, resulting in nonoptimal scaling.

By minimally processing, we mean that we simply identify the two-word clusters that occur frequently within a particular time interval in the stream, and record the occurrence counts for these two-word clusters across time intervals in a database. This is straightforward. The novelty of our approach lies in our heuristic algorithm for computing multi-word clusters across multiple time intervals using these base two-word clusters. The heuristic algorithm has four characteristics.

— The heuristic provides very high recall, i.e., nearly all high-frequency multiword clusters will be identified by our heuristic.
— The heuristic provides good precision when the number of words required to denote the topic is relatively low (up to 5–6 words). Given that microblog posts are limited to 140 characters, and not all words in a post will be required to characterize a trending topic, this is more than sufficient for trending topic analysis in microblogs.
— The algorithm filters out word clusters that frequently occur, since these word clusters represent known important topics. We are particularly interested in the word clusters for which there is a sudden upward change in the frequency. For this, we develop a measure of burstiness as a filtering mechanism.
— The algorithm is fast and scalable, with online responsiveness under heavy post input and query arrival rates.

Table II. Notation

Notation	Meaning
S^i	word cluster set consisting of word clusters each of size i
S	$\cup_{i>=2} S^i$
C^i	word cluster consisting of i words
C	$\cup_{i>=2} C^i$
C^i_j	j^{th} word cluster consisting of i words
W_{jn}	the n^{th} word in cluster j
$threshold_{support}$	user specified threshold value for support when identifying frequent word clusters
$threshold_{burstiness}$	user specified threshold value for burstiness
$Sup(A,B)$	support of words A and B
$B_{T,C}$	burstiness of cluster C in time period T
$UStartTime$	start time of user entered query time range
$UEndTime$	end time of user entered query time range
$<Table>.<col>$	field value 'col' of table 'Table"
w	number of words in a post
c	average number of words in a cluster repressing a trending topic
L	maximum number of frequent clusters
r	number of posts received per second
$\rho_{<Q>}$	standard Traffic Intensity in a queue Q
$\lambda_{<Q>}$	poisson arrival rate queue elements of queue Q

In this research, we make two specific contributions:

(1) We develop a heuristic to identify frequently occurring multiword clusters based on two-word item-sets.
(2) We develop a measure of burstiness for these frequently occurring multiword clusters to identify trending topics.

Before detailing our approach, we first define the notation we will use throughout the remainder of this article in Table II.

4. SOLUTION DETAILS

In this section, we describe the architecture of our *TrendMiner* system, and the details of each component in the architecture.

4.1. Architecture

At a high level, *TrendMiner* consists of two parts: (a) an input text stream handler, and (b) an online query processor, as depicted in Figure 1. The input stream handler is responsible for processing the incoming stream of posts and identifying the frequency of word-pairs occurring within a time interval. The online query processor is responsible for accepting user queries for trending topics within a set time interval, identifying the frequently-occurring multiword clusters from the stored two-word clusters, and identifying the burstiness of frequently-occurring clusters. We describe each of these modules in detail.

4.2. Input Text Stream Handler

The input text stream handler takes as input, a stream of posts containing text and processes them through a set of modules: a *Receiver*, a *Parser*, a *Processor*, an *In-memory Hash Table Data Store*, an *Updater*, and a *Database*. We describe each of these modules in the following.

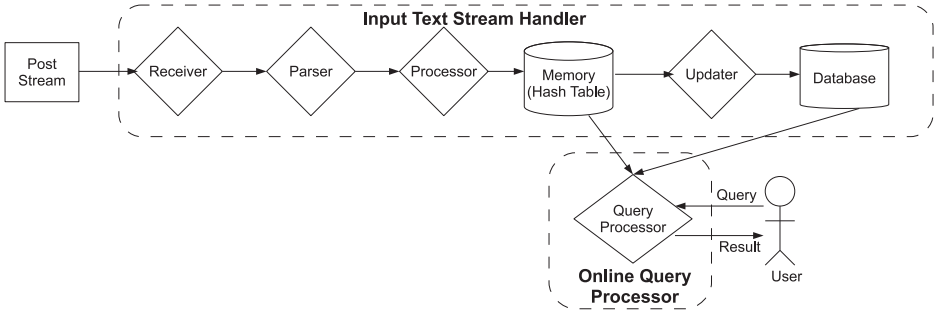

Fig. 1. *TrendMiner* architecture.

Receiver. Within the stream handler, the Receiver component receives the text of a post to the stream, and runs a heuristic to identify whether the text of the post is written in English. (It is possible to extend our approach for other languages using different dictionaries, however, for this research we focus only on English posts.)

The heuristic to identify whether a post is written in English is based on a scoring metric that considers (1) the number of English words in the post that match words in a well-defined English dictionary, (2) the origin of the post (posts from English-speaking countries are more likely to be in English), and (3) the language settings of the person who posted it. First, we compute the percentage of valid English words out of the total number of words in the post. Next, based on the origin of the post and the language settings of the person who posted it, a threshold percentage value is selected for the number of English words required in the posted text in order to consider it to be an English-language post. If the origin of the post is an English-speaking country and the language setting is found to be English, then the threshold percentage is set at a lower value than if the origin is a not associated with English-language speakers, or if the language setting is not English. The exact values for the thresholds were determined by experimental observation. For posts with an English-speaking origin and English as the language setting, the threshold value is 50%. If one of them is non-English, then the threshold value is 80%. If both of them are non-English, the threshold value is set at 90%. If the post is in English, the Receiver passes the text of the post to the Parser.

Parser. The Parser divides the content of posts into sentences, and the sentences into words. It then assigns part-of-speech tags to those words by applying OpenNLP.[1] The parser eliminates stop words (e.g., "a," "the," "I," "we"), as well as any word that is not a noun or a verb. The parser then stems each word to its canonical form. The result of the parser step is a document word list of all the words that might be relevant for trending topic analysis. To demonstrate the result of the parsing step, consider the following sample post.

Rupert Murdoch attacked in Parliament during testimony.

For this post, the parser generates the following list: "Rupert," "Murdoch," "attack," "Parliament," "testimony." Note that stemming reduces "attacked" to "attack," and part-of-speech filtering removes the prepositions "in" and "during." When the Parser is finished, it places the resulting word-list in a queue for the Processor, which we describe next.

[1]http://incubator.apache.org/opennlp/

Table III. TimeID Table

Tid	time id of update period
StartTime	start TimeStamp of an update period
EndTime	end TimeStamp of an update period
DocFreq	number of documents processed in that update period

Table IV. Cooccurrence Table

ID	unique number to represent (WordId_1, WordId_2)
WordId_1	1st WordId of the cluster (WordId_1, WordId_2)
WordId_2	2nd WordId of the cluster (WordId_1, WordId_2)
Support	Sup(WordId_1, WordId_2)
Tid	time id of update period

Processor. The Processor module fetches each word-list from the parser's result queue and generates all pairs of cooccurring words. For instance, in the context of our example post, the processor would generate word-pairs including ("Rupert," "Murdoch"), ("Rupert," "attack") and several others.

In-memory Hash Table Data Store. When the Processor is finished creating cooccurring word pairs, it then updates cooccurrence information for each word-pair in a memory-resident hash table.

Database. A database stores all the frequently occurring word-pairs in each time interval. The database consists of two tables, `TimeID` (Table III) and `Cooccurrence` (Table IV).

The `TimeID` table stores the start and end times for each trending topic analysis interval, as well as a count of the documents processed in the period. The `Cooccurrence` table stores frequently-occurring word-clusters within each time interval.

Updater. The Updater runs at regular intervals and computes the support, $Sup(W_i, W_j)$, for each word-pair, (W_i, W_j), within that interval. Conceptually, $Sup(W_i, W_j)$ is the probability that two words W_i and W_j occur together in a post. We define $Sup(W_i, W_j)$ as follows.

$$Sup\left(W_i, W_j\right) = \frac{|D_{ij}|}{|D|}, \tag{1}$$

where D_{ij} is the set of documents within that interval that contain both the words W_i and W_j, and D is the total set of documents within that interval. The Updater then identifies all the word pairs that meet a minimum threshold support value within this interval. For instance, the word pair ("Rupert," "Murdoch") would likely meet the threshold.

Finally, the updater updates the database tables `Cooccurrence` (Table IV) and `TimeID` (Table III) with the word pairs and the time interval details respectively. In the `Cooccurrence` table, the WordId for a word is a unique integer identifier for the word (we use the integer identifiers associated with words in the Wordnet corpus for this purpose).

Having described input stream processing, we now move on to the online query processing module.

4.3. Online Query Processing

In this section, we describe our heuristics for identifying trending topics within a user-specified time range [*UStartTime*, *UEndTime*], i.e., identifying the word clusters that have relatively higher frequency within the specified time range than in earlier time periods.

We first describe our heuristic for identifying frequently-occurring word clusters of 3 or more words from the frequently occurring two-word clusters in the `Cooccurrence` table. We then describe how we use this estimated frequency of each cluster of 3 or more words to measure the burstiness of word clusters to identify trending topics. We describe each of these steps in detail.

4.3.1. Identifying Frequent Word Clusters in a Time Range. This heuristic for identifying frequent word clusters in a time range is composed of the following steps.

(1) Identify the relevant update periods in the `TimeID` table (Table III) that cover the time range [*UStartTime*, *UEndTime*] in the user query.
(2) Identify frequently occurring two-word clusters within the time range [*UStartTime*, *UEndTime*].
(3) Generate larger word clusters from the frequently occurring two-word clusters as identified in Step 2.

Algorithm 1 summarizes the implementation of these steps; we describe the details of each step in the following.

ALGORITHM 1: Finding Word Clusters

Input: Time period defined by [*UStartTime*, *UEndTime*]
Output: Frequently occurring cluster set $\mathcal{S}$

1 From the table `TimeID`, identify set of all update periods $\mathcal{T}$, that covers the time period [*UStartTime*, *UEndTime*]
2 **forall the** *record* $\in$ `Cooccurrence` *where record.tid* $\in \mathcal{T}$ **do**
3 | Compute $Sup(record.WordId_1, record.WordId_2)$ following Equation (1)
4 | **if** $Sup(record.WordId_1, record.WordId_2)) > threshold_{support}$ **then**
5 | | $S^2 \leftarrow S^2 \cup \{(record.WordId_1, record.WordId_2)\}$
6 | **end**
7 **end**
8 **for** $i = 2; S^i \neq \emptyset\ ; i++$ **do**
9 | **forall the** $C^i_j, C^i_k \in S^i; j \neq k$ **do**
10 | | $C^{i+1}_{new} \leftarrow ClusterGeneration(C^i_j, C^i_k)$
11 | | **if** $C^{i+1}_{new} \neq \emptyset$ **then**
12 | | | $S^{i+1} \leftarrow S^{i+1} \cup \{C^{i+1}_{new}\}$
13 | | **end**
14 | **end**
15 | $\mathcal{S} \leftarrow \mathcal{S} \cup S^{i+1}$
16 **end**
17 **forall the** $C^i \in \mathcal{S}$, $i >= 2$ **do**
18 | **if** $B_{[UStartTime,UEndTime],C^i} < threshold_{burstiness})$ **then**
19 | | Remove C^i from $\mathcal{S}$
20 | **end**
21 **end**
22 **return** $\mathcal{S}$

Identify the update periods. To find the update periods of interest, we first query the database to find the update periods whose end times are after the start of the query time range *UStartTime* and whose start times are before the end of the query time range *UEndTime* (Algorithm 1, line 1).

Identify frequently occurring two-word clusters. This step involves first identifying the frequently occurring two-word clusters in each of the update periods identified in

Step 1. These word-pairs reside in the Cooccurrence table (Table IV). For each update period in the query time range [*UStartTime*, *UEndTime*], we draw word pairs ($WordId_1, WordId_2$) from the database Cooccurrence table and compute the support, $Sup(WordId_1, WordId_2)$, as given in Equation (1) (Algorithm 1, line 3).

Next, we compute the total frequency of each two-word cluster within the time range [*UStartTime*, *UEndTime*] by summing up the frequencies found from the database for each relevant update period. We compute the support for each two-word cluster using Equation (1), and retain only those two-word clusters that meet a minimum support threshold. We check whether the $Sup(WordId_1, WordId_2)$ of the fetched word-pair is greater than a configurable $threshold_{support}$ (Algorithm 1, line 4). If it is, the word-pair is added into a set of two-word clusters denoted by S^2 (Algorithm 1, line 5).

Generate the larger word clusters. In Algorithm 1, lines 8–15, we start with an initial set of 2-word clusters (S^2). We then merge clusters containing common words to form larger clusters. Generically, by merging two i-word clusters, we generate a single $(i+1)$-word cluster (Algorithm 1, line 10). The details of the merging algorithm are given in Algorithm 2.

In Algorithm 2, we merge two i-word clusters if they have $i-1$ common words. In order to accelerate this process, we order each cluster based on the WordID of each word. If words in the clusters are ordered, comparing two i-word clusters is a matter of checking that the first $i-1$ words of each cluster match, i.e., the two clusters C_j^i and C_k^i can be merged only if $W_{jn} = W_{kn}, \forall n = 1, 2, ..., i-1$. (Algorithm 2, lines 4–5).

If $W_{jn} \neq W_{kn}$ for any value of n, we stop checking and return a null set. If $W_{jn} = W_{kn}$ for all values of n ($n = 1, 2, ..., i-1$), we merge the clusters to generate a new $(i+1)$-word cluster C^{i+1} (Algorithm 2, lines 12 and 15).

When merging clusters, we append the word with the higher WordId to the end of the cluster that does not include that word in order to maintain ordering by increasing WordID in the cluster (Algorithm 2, lines 11–12 and 14–15). The size of the clusters produced by our method depends on the input, and how much wording variation occurs in posts describing the same trend. In practice, the size of the cluster is typically between two and six words.

We illustrate this cluster-merging process with an example. Consider two clusters $\mathcal{C}$: ("Rupert," "Murdoch," "attack") and $\mathcal{C}'$: ("Rupert," "Murdoch," "testimony"), in which words are in increasing WordId order. If the WordId of "attack" is greater than "testimony," we append "attack" to the end of $\mathcal{C}'$, such that the new $\mathcal{C}'$ contains ("Rupert," "Murdoch," "testimony," "attack"); otherwise, "testimony" is appended to the end of $\mathcal{C}$, such that the new $\mathcal{C}$ contains ("Rupert," "Murdoch," "attack," "testimony").

In merging the two clusters, we want to retain only those clusters that are themselves frequent, i.e., whose *Support* value meets the minimum $threshold_{support}$ value. To do this, we only need to check the support of the two nonintersecting words from the original clusters (Algorithm 2, line 10). For example, if we have two word clusters ("Rupert," "Murdoch," "attack") and ("Rupert," "Murdoch," "testimony"), we only need to check the support of "attack" and "testimony," because other pairs of words, namely ("Rupert," "Murdoch"), ("Rupert," "attack"), ("Murdoch," "attack"), ("Rupert," "testimony"), ("Murdoch," "testimony"), have already appeared in one cluster, indicating that their support values are greater than the $threshold_{support}$.

PROPOSITION 4.1. *The merging method of generating $i+1$ word clusters from two i-word clusters is guaranteed to capture frequently occurring clusters without false negatives.*

ALGORITHM 2: ClusterGeneration

Input: Two clusters C_j^i, C_k^i of size i

Output: New word cluster C^{i+1} of size $i+1$ or $\emptyset$

```
Let W_jn be the n^th word in the cluster C_j^i
Let W_kn be the n^th word in the cluster C_k^i
Initialize flag ← true
forall the W_jn ∈ C_j^i and W_kn ∈ C_k^i, n ≤ i − 1 do
    if W_jn ≠ W_kn then
        flag ← false
        break
    end
end
if flag = true and Sup(W_ji, W_ki) ≥ threshold_support then
    if W_ji < W_ki then
        C^{i+1} ← Append W_ki to C_j^i
    end
    else
        C^{i+1} ← Append W_ji to C_k^i
    end
    return C^{i+1}
end
else
    return ∅
end
```

For an i-word cluster to be frequent, its constituent $(i-1)$-word clusters must also meet the frequency $threshold_{support}$. This applies recursively until the base case of $(i-1)=2$. Since we are starting with S^2, where all the 2-word clusters in S^2 meet the frequency criteria, and the method considers all possible pairings of clusters in S^2, we can guarantee that all frequent 3-word clusters are included in S^3. Further, we can be sure that all frequent 4, 5, ..., i-word clusters are included in S^4, S^5, ..., S^i, respectively.

We demonstrate the proposition with an example. Consider the cluster ("Rupert," "Murdoch," "attack," "testimony"), which is ordered by WordID, as an example. Here, we know that the two subclusters ("Rupert," "Murdoch," "attack") and ("Rupert," "Murdoch," "testimony") must be included in frequent 3-word clusters, since they are also frequent clusters. By checking the first two words of ("Rupert," "Murdoch," "attack") and ("Rupert," "Murdoch," "testimony"), we can generate the 4-word cluster ("Rupert," "Murdoch," "attack," "testimony"). Of course, there are other pairs of 3-word subclusters, say ("Rupert," "attack," "testimony") and ("Murdoch," "attack," "testimony"). The checking process of this pair will stop immediately when it finds that "Rupert" is not equivalent to "Murdoch". There is no need to check these other pairs, since this would generate the same 4-word cluster considered by our strategy of combining based on final words in clusters. By short-circuiting processing of redundant clusters, we can save significant effort and time, without losing any frequent clusters.

While we guarantee that our method will not exclude frequent word-sets, it is possible that false positives may be generated. Our method takes a bottom-up approach, assuming that an $(i+1)$-word cluster that is generated from two frequent i-word clusters is also frequent. This may not be the case. Consider, for example, a scenario in which all three of the word-pairs (A,B), (B,C), and (A,C) meet the $threshold_{support}$.

Our method will consider the 3-word cluster (A,B,C) to be frequent, even though those three words may not actually cooccur frequently enough in the set of input posts to meet $threshold_{support}$.

4.3.2. Identifying Trending Clusters. At this point, we have identified all frequent clusters in the user-specified time range [$UStartTime, UEndTime$]. However, not all of them represent trending topics. These clusters fall into two classes: clusters that are regularly frequent (these clusters typically exceed $threshold_{support}$ for most time ranges); and clusters that are more frequent in the specified time range than they are on average, i.e., they are trending. We wish to identify clusters in the latter case.

Again, following the lead of existing work [Glance et al. 2004; Liang et al. 2010], we use the concept of *burstiness* to measure the degree of trending of clusters. We define a bursty cluster as follows.

Definition 4.2. A *bursty cluster* is one that exhibits an unusually high frequency over a finite time window compared to its past occurrences. In other words, if there is a significant increase in frequency over a time interval, we consider this cluster to be bursty and trending in that time interval.

We model the average frequency of a word cluster C at the time t $(\bar{f}_{t,C})$ as the average of the frequencies for the word cluster C $(f_{n,C})$ across t_{prev} update periods before t. Thus,

$$\bar{f}_{t,C} = \frac{1}{t_{prev}} \sum_{n=(t-1-t_{prev})}^{t-1} f_{n,c}, \tag{2}$$

where $f_{n,C}$ is the frequency of C in the n^{th} update period.

Using this average frequency, we model the burstiness of a word cluster C at the time t as the ratio of the actual frequency and the expected frequency in a time interval. Thus we define

$$B_{t,C} = \frac{f_{t,C}}{\bar{f}_{t,C}}, \tag{3}$$

where $B_{t,C}$ is the burstiness of the word cluster C in the time interval t.

One of the challenges in measuring burstiness is knowing the frequency of each frequent word cluster C for each update period. We only store 2-word clusters and their actual frequencies for each update period in the `Cooccurrence` table. Thus, we developed a heuristic to estimate the frequency of any frequent word clusters C, identified in lines 8–16 of Algorithm 1.

We use the minimum value of the frequencies of all possible 2-word clusters (W_j, W_k) within a larger cluster C as the approximate cluster frequency for C. Thus formally,

$$f_{t,C} = min_{\forall (W_j,W_k):W_j \in C \& W_k \in C} f_{t,(W_j,W_k)}, \tag{4}$$

where $f_{t,(W_j,W_k)}$ is the frequency for the two word cluster (W_j, W_k) at update period t as stored in the database table `Cooccurrence`.

Following the measure of burstiness in Equation (3), we use the the frequency defined by Equation (4) to compute burstiness for each word cluster $C \in S$ for the user-specified time range [$UStartTime, UEndTime$] in Algorithm 1, line 18, as follows.

$$B_{[UStartTime,UEndTime],C} = \frac{\bar{f}_{[UStartTime,UEndTime],C}}{\bar{f}_{UStartTime,C}}, \tag{5}$$

where $\bar{f}_{[UStartTime,UEndTime],C}$ is the average frequency per update period within the time range [$UStartTime, UEndTime$].

Thus the burstiness within a query's time range [*UStartTime*, *UEndTime*] is computed as the ratio of the average frequency per update period within the queried time range and the average frequency per update period for t_{prev} update periods before the query's time range.

If the burstiness for the word cluster C within the queried time range meets a predefined threshold value $threshold_{burstiness}$, the word cluster is kept in the trending set S; otherwise, it is removed from S (Line 19, Algorithm 1).

At the end, in line 22 of Algorithm 1, the set S of word clusters is returned. Each word cluster $C \in S$ defines a trending topic for the user-queried time range [*UStartTime*, *UEndTime*].

5. COMPLEXITY ANALYSIS

In this section, we discuss the time and space complexity of *TrendMiner*.

5.1. Time Complexity

TrendMiner consists of two different types of processing: handling the input text stream and processing online user queries for trending topic detection. Since these are two independent processes, we demonstrate the time complexity of each of these separately.

5.1.1. Input Text Stream Handler. Let us assume that a post has w words. As described in Section 4, first the Receiver receives a post and places it in a buffer queue, which takes $O(1)$ time. Next, the Parser parses the post into w words, which takes w units of time. After that, the Processor generates 2-word clusters from the w words, which contributes $\binom{w}{2}$ unit steps, i.e., $O\left(w^2\right)$. Next, the Updater transfers frequent 2-word clusters from the hash table to the database. Without loss of generality, let us assume that all 2-word clusters are frequent. Thus, appending a $\binom{w}{2}$-word cluster takes $O\left(w^2\right)$ insertions in the database. Therefore, the time complexity to process one post can be expressed as $\left(O(1) + w + O\left(w^2\right)\right) = O\left(w^2\right)$, which is quadratic to the number of words in the incoming post.

5.1.2. Online Query Processing for Finding Trending Clusters. To find the trending clusters, we first need to find frequent clusters (meeting $threshold_{support}$) and then find the bursty clusters (meeting $threshold_{burstiness}$). To find the frequent clusters, we need to access the cooccurrence information from the database and fetch the 2-word clusters corresponding to the update time periods within the query time range. Let us assume that there are M update periods within the query time range. Thus, it will require M database fetches to gather the 2-word clusters for the M update periods.

The time required to generate an $(i + 1)$-word cluster from two i-word clusters is $O(i)$. Thus, assuming that there can be at most L clusters and each cluster can have maximum c words, the time complexity required to find frequent clusters would be,

$$O\left(\sum_{i=3}^{c}\binom{L}{2} \times i\right) = O\left(c^2 \cdot L^2\right). \tag{6}$$

Next, to find the bursty clusters in the current time period, we need to find the average frequency of all clusters in the previous t_{prev} update periods and the M update periods within the query time range [*UStartTime*, *UEndTime*]. Assuming at most L clusters, where each cluster has a maximum of c words, the number of database operations required to find the average frequencies for each cluster across all $t_{prev} + M$

update periods is $O\left(\binom{c}{2} \times L\right)$. Assuming the database operations take constant time, the total time complexity in computing burstiness for L clusters would be,

$$O\left(c^2 \cdot L \cdot (t_{prev} + M)\right). \tag{7}$$

Thus, the total time complexity for processing the input query to find trending topics/clusters would be,

$$O\left(M + c^2 \cdot L^2 + c^2 \cdot L \cdot \left(t_{prev} + M\right)\right). \tag{8}$$

PROPOSITION 5.1. *The time complexity of processing the input query is quadratically related to both L and c.*

Note that the value of c is determined by an external system such as Twitter (the 140-character limit for microblog posts effectively limits c to a small integer value). The value of L can be controlled by the value of the parameter $threshold_{support}$. If the value of $threshold_{support}$ is lower, the value of L would be smaller and would reduce the overall time complexity of the system. On the other hand, a very low value for $threshold_{support}$ can omit some trending clusters in the final output. Therefore, the value of the $threshold_{support}$ needs to be judiciously chosen based on available system resources and the desired sensitivity of the results.

PROPOSITION 5.2. *The time complexity of processing the input query is linearly related to M.*

If the query time range [*UStartTime*, *UEndTime*] is wider, the value of M would be larger. So, from Equation (8), we can find that the time complexity of the system would increase linearly along with the query time range.

5.2. Storage Complexity

The *Updater* module periodically transfers word cooccurrence information from the memory-based hash table to the database. The memory consumed by the 2-word clusters generated within an update period is flushed regularly, and thus does not grow over time. However, the database maintains historic data for word cooccurrence across update periods, and thus does grow over time. Therefore, we estimate the bound of the database size in this analysis.

At the end of each update period, the contents in the memory-based hash table are transferred to the database tables `Coocccurrence` and `TimeID`.

The content of `TimeID` is directly dependent on how much historical data we want to store in the database and update period. For shorter update periods, the size of the table `TimeID` also increases linearly. As the length of the history grows, the size of the `TimeID` table also increases linearly. Each row of the `TimeID` table is of size $32 + 64 \times 2 +$ 32 bits = 24 bytes. If the update period is 10 minutes, i.e., at every 10-minute interval the 2-word clusters and their frequencies are transferred from the memory-resident hash table to the database `Cooccurrence` table, and we keep a history of one week, the total size of the data in the `TimeID` would be, $24 \times 6 \times 7 \times$ 24 bytes = 24.2 kB.

Each row of the `Cooccurrence` table has a size of $32 + 32 + 32 + 32 +$ 32 bits = 20 bytes. Let us assume that posts arrive at an average rate of r posts per second, with each post consisting of w words on average. Then each post will create $\binom{w}{2}$ 2-word clusters. Assuming only $p\%$ of the 2-word clusters meet the threshold support requirement, $\binom{w}{2} \times \frac{p}{100} \times r \times 20 \times$ 600 bytes are added to the database at each update period. In March 2011, Twitter received on average, 1620 posts per second [Twitter 2011], i.e.,

$r = 1620$. Each Twitter post is at most 140 characters in length. Assuming each word is on average 6 characters long, we can assume the Twitter post will have on average 24 words. Assuming $p = 5$, i.e. only 5% of the 2-word clusters cross the support threshold within each update period, then at each update period the database size grows by 268 MB. If we keep one week's historical data, the data size for the `Cooccurrence` table would be $268 \times 6 \times 24 \times 7$ MB = 270 GB, which is manageable considering the scale of the problem we are handling.

6. SCALABILITY ANALYSIS

In this section we demonstrate how the system scales based on a queuing theory model.

TrendMiner can be modeled as two independent queues: Q_1 for the input text stream handler, and Q_2 for the query handler. The input for Q_1 is incoming posts. The input for Q_2 is a stream of user queries requesting trending topics within specific intervals. Average processing time, service time, and traffic intensity have their usual meanings in queueing theory.

For Q_1, we assume a M/D/1 model, i.e., we assume that the distribution for the interarrival rate of posts follows a Poisson distribution with mean λ_{Q_1}. The processing time for an incoming post is deterministic, and follows from our time complexity computation.

Let q_m be the average number of posts in the system at the moment when the processing of the m^{th} post finishes. Suppose that γ_m is the number of posts that arrive during the service time of post m. Let ρ_1 be the traffic intensity for the queue Q_1. Thus, we have the expected number of posts in the system

$$E[q_m] = \rho_1 + \frac{\rho_1^2}{2(1-\rho_1)} = \frac{\rho_1}{1-\rho_1}\left[1 - \frac{1}{2}\rho_1\right]. \tag{9}$$

The average waiting time β_1 in the queue Q_1 will be given by

$$\begin{aligned} \beta_1 &= \frac{1}{\lambda_{Q_1}} \frac{\rho_1^2}{2(1-\rho_1)} \\ &= \frac{\lambda_{Q_1} c^2}{1-\lambda c} \\ &= O\left(\frac{\lambda_{Q_1}}{4} \frac{\left(w^2+w+2\right)^2}{2-\lambda_{Q_1}\left(w^2+w+2\right)}\right). \end{aligned} \tag{10}$$

For Q_2, we assume a M/D/1 model, i.e., we assume that users submit queries at a random rate. These queries are processed by the query handler in deterministic time as described by our time complexity analysis. As we did in the case of Q_1, here we also assume a single thread (this can easily be extended to multiple threads). Let us denote the query of Q_1 as κ_1. Without loss of generality let us assume that the arrival rate of queries is λ_1. Let the arrival rate follow the Poisson distribution function. From the complexity analysis, the time needed to process the query is $O(M + c^2 \cdot L^2 + c^2 \cdot L \cdot (t_{prev} + M))$.

The traffic intensity for the queue Q_2 is given by

$$\rho_2 = \lambda_1 O\left(M + c^2 \cdot L^2 + c^2 \cdot L \cdot (t_{prev} + M)\right). \tag{11}$$

The average waiting time for queue Q_2 is given by

$$\begin{aligned}\beta_2 &= \frac{1}{\lambda_1}\frac{\rho_2^2}{2(1-\rho_2)} \\ &= O\left(\frac{\lambda_1}{4}\frac{\left(M + c^2 \cdot L^2 + c^2 \cdot L \cdot (t_{prev} + M)\right)}{2 - \lambda_{Q_2}\left(M + c^2 \cdot L^2 + c^2 \cdot L \cdot (t_{prev} + M)\right)}\right).\end{aligned} \tag{12}$$

Hence the total waiting time of the system is given by the combined waiting time for Q_1 and Q_2.

$$\begin{aligned}\beta_1 + \beta_2 &= O\left(\frac{\lambda_{Q_1}}{4}\frac{(w^2 + w + 2)^2}{2 - (w^2 + w + 2)}\right) \\ &\quad + O\left(\frac{\lambda_{Q_2}}{4}\frac{M + c^2 \cdot L^2 + c^2 \cdot L \cdot (t_{prev} + M)}{2 - \left(M + c^2 \cdot L^2 + c^2 \cdot L \cdot (t_{prev} + M)\right)}\right).\end{aligned} \tag{13}$$

Therefore, the average waiting time from the time a post has arrived at the Input Text Handler to the time that it can be recognized by a query in identifying trending topics is given by

$$O\left(\lambda_{Q_1} w^2\right) + O\left(\lambda_{Q_2}\right), \tag{14}$$

where λ_{Q_1} is the average interarrival rate for a post, λ_{Q_2} is the average interarrival rate for query, and w is the average number of words in a post.

PROPOSITION 6.1. *The average waiting time for each incoming user query in TrendMiner is quadratic to the average number of words in a twitter post (w).*

Due to the linear-quadratic nature of Equation (14), *TrendMiner* is scalable. In the next section, we demonstrate this scalability experimentally.

7. EXPERIMENTAL RESULTS

In this section, we experimentally demonstrate the quality and scalability of our approach.

7.1. Experimental Environment

We developed an experimental testbed based on the architecture depicted in Figure 1 and the algorithms described in Section 4. All modules, including not only the *TrendMiner* modules, but also all helper software and all modules representing comparison approaches, were implemented in Java 1.6. A MySQL v5.1 DBMS was installed to store the 2-word clusters and their frequencies. All modules and the database were installed in the same 2.33 GHz quad-core CPU, 8 GB RAM server running Linux Ubuntu 11.04.

For the purpose of conducting the experiments, we collected a set of approximately one million Twitter posts to serve as test input data; we call this dataset the *Twitter Post Bank* (TPB). Further, we implemented two helper modules: (a) a *Text Stream Generator* (TSG), and a *Query Generator* (QG). The TSG extracts posts from the TPB and submits them to the system as a Poisson process with a configurable mean interarrival time, *Post Arrival Rate*. The QG formulates queries for trending topics of the form: *retrieve all trending clusters in the interval* [*UStartTime*, *UEndTime*], where a cluster is a set of words representing a trending topic, and submits them to the system. To formulate such a query, the QG first fixes the interval of interest by generating

Table V. Baseline Experimental Values

Parameter Name	Baseline Value	Range
$threshold_{support}$	0.015	N/A
$threshold_{burstiness}$	1.5	N/A
Update Period	10 minutes	N/A
Post Arrival Rate	300 posts/second	100-900
Query Arrival Rate	10 queries/second	5-30

the *UStartTime* and *UEndTime* values by drawing both values from a uniform distribution delimited by *[experiment begin time, current time]*. Two additional constraints are imposed: first, *UEndTime* must be greater than *UStartTime* and second, 60% of the queries are forced to have *UEndTime* set to current time to simulate the detection of current trending topics. Obviously, the remaining 40% are historical queries for trending topics. Table V shows the baseline values used in our experiments.

All reported values were obtained by running each experiment ten times and reporting the average across all experimental runs for each reported data point.

7.2. Quality of Results

In this section, we present a set of experimental results describing the quality of results provided by *TrendMiner*.

We first present the results of a comparison of trending topics from Twitter and trending topics detected by *TrendMiner*. Due to the limited amount of data available from Twitter, these results are necessarily both high-level and qualitative. Currently, Twitter only provides a maximum of 1% of the total Twitter feeds within a given time. Within a short interval (such as a few minutes) this 1% of Twitter posts can hardly be used to identify trending topics any better than Twitter's current trending topic feature. Thus, we have focused on the daily feeds and have qualitatively compared the results of our trending topic identification method with the Twitter trending topics.

To provide a more nuanced sense of the quality of the *TrendMiner* results, we present the results of a set of accuracy experiments that compare the *TrendMiner* results to those discovered by an exhaustive method guaranteed to find all trending topics meeting $threshold_{support}$ and $threshold_{burstiness}$. In the exhaustive method, the 2, 3.. n-word clusters are generated by scanning the complete post database. Since there is no approximation employed in this method, it produces all possible clusters crossing the respective thresholds. This method serves here as a gold-standard of accuracy; however, since it requires multiple database scans, it is not appropriate for use in online scenarios.

7.2.1. Comparison with Twitter Trending Topics. Each day, Twitter publicly releases a subset of daily posts. For a week (from 22 February 2012 to 28 February 2012), we ran these posts (roughly 12,000 per day) through *TrendMiner* to identify the trending word clusters each day. For the same week, we recorded Twitter's daily top five trending topics, and manually mapped the top-five Twitter trending topics to the *TrendMiner*-identified trending word clusters. We report the corresponding Twitter and *TrendMiner* trending topics in Table VI.

TrendMiner is clearly able to identify trending topics in the post stream. For example, the release of a new movie, "Jump Street," was identified as the 13th trending topic, the news of the death of actor Erland Josephson at the age of 88 came up as the 22nd trending topic. Of the 35 trending topics identified by Twitter, we were able to map 33 to the trending word clusters identified by *TrendMiner*; specifically, items 8 and 9 were not mapped. This is due to the fact that *TrendMiner* currently only considers English-language posts; any post that does not contain sufficient English words is

Table VI. Comparison with Twitter Result

Twitter Trending Topics vs. TrendMiner Trending Topics			
NO.	**Twitter Trending Topics**	**TrendMiner**	**Trending Date**
1	Ash Wednesday	[ash, wednesday, lenten, start, today]	22 Feb. 2012
2	lent	[ash, wednesday, lenten, start, today]	
3	TITANIC in 3D	[titanic, april, 3d, watch]	
4	Miss St.	[defense, bost, dee, state, gilchrist, kidd, st, mississippi]	
5	Mississippi State	[defense, bost, dee, state, gilchrist, kidd, st, mississippi]	
6	Keep Quinn Walking	[walking, keep, magick, spread, quinn]	23 Feb. 2012
7	Ash Wednesday	[ash, wednesday, lenten, start]	
8	El Del Arte Malicioso		
9	By PDN		
10	Jason Terry	[product, jeremy, system, terry, lin, antoni, jason]	
11	Happy Birthday Steve Jobs	[steve, jobs, birthday]	24 Feb. 2012
12	Brandon Jacobs	[wrestling, jacobs, tna, brandon, giants, impact]	
13	Jump Street	[21, screening, street, movie, jump]	
14	Gregg Jevin	[rip, die, gregg, jevin]	
15	Estelle Silvia Eva Mary	[princess,victoria, name, estelle, silvia, eva, daughter, mary]	
16	RIP Facebook	[rip, facebook, die]	25 Feb. 2012
17	Tahs	[rebels, tahs, mortlock, team, clash]	
18	Congrats DLSU	[season, congrat, volleyball, dlsu]	
19	Tom and Siva	[tom, siva, cry]	
20	Dom Shipperley	[reds, dom, shipperley, win]	
21	R.I.P Rowan Atkinson	[bean, hoax, death, rowan, atkinson, r.i.p]	26 Feb. 2012
22	Erland Josephson	[erland, die, age, josephson, 88]	
23	Samsung Galaxy Beam	[announce, beam, samsung, galaxy, projector]	
24	Scott Parker	[scott, england, parker, player, vote, year]	
25	Arsenal 5-2 Tottenham	[arsenal, beat, tottenham]	
26	Bret McKenzie	[mckenzie, bret, oscar, win]	27 Feb. 2012
27	The Iron Lady	[meryl, streep, lady, win, oscar, iron]	
28	Meryl Streep	[meryl, streep, lady, win, oscar, iron]	
29	Mahamadou Diarra	[sign, fulham, diarra, mahamadou]	
30	Photoshop Touch	[launch, ipad, adobe, photoshop, touch]	
31	Jaleel White	[jaleel, dance, cast, star]	28 Feb. 2012
32	Dancing with the Stars	[cast, donald, driver, join, dance, star]	
33	Donald Driver	[cast, donald, driver, join, dance, star]	
34	Lucy Liu	[lucy, liu, cast, watson]	
35	Juan Pablo Montoya	[juan, pablo, montoya, crash, dryer, daytona, jet, 500]	

omitted from consideration. This can be changed easily by adding a language identifier and respective language-specific logic in the *Parser* module depicted in Figure 1.

A comparison of *TrendMiner* trending topics and Twitter trending topics shows that *TrendMiner* is able to identify trending topics with higher contextual content than Twitter trending topics. For example, on 25 February, Twitter identified "Congrats DLSU" as a trending topic. If one is not familiar with DLSU, this is not a very meaningful trending topic. In contrast, *TrendMiner* was able to identify the trending topic as congratulating the DLSU volleyball team. In another example, Twitter reported "RIP Rowan Atkinson," while the additional context provided by *TrendMiner* was able to identify the trending topic as actually a hoax.

A further inspection of Table VI shows that the higher context provided in *TrendMiner* trending topics helps to identify single trending topics that Twitter identifies under multiple separate trending topic labels. For example, Twitter identified "Meryl

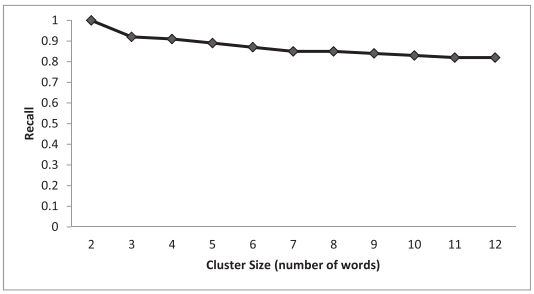

Fig. 2. Recall.

Streep" and "Iron Lady" as separate trending topics on 27 February, where *TrendMiner* identified them as part of the same trending topic, and provided the additional context of the actress' Oscar win.

7.2.2. Accuracy. In this section, we aim to measure how the approximation affects the recall and precision of our method with the increase of cluster size. As there is no approximation in the exhaustive method, to demonstrate accuracy, we compare results obtained by our methods to those obtained by running the exhaustive method, which is guaranteed to produce a set of optimal clusters.

To quantify accuracy, we use the traditional *precision* and *recall* metrics, defined as follows (where $Result_{TM}$ represents the results obtained from our *TrendMiner* method, and $Result_E$ represents the results obtained from the exhaustive method):

$$Recall = \frac{|Result_{TM} \bigcap Result_E|}{|Result_E|}$$

$$Precision = \frac{|Result_{TM} \bigcap Result_E|}{|Result_{TM}|}.$$

These are reported in Figures 2 and 3, where we report recall and precision, respectively, for varying cluster sizes.

Figure 2 shows our recall results. Clearly, our methods demonstrate high recall accuracy, ranging from 1.0 for 2-word clusters (for which we have actual frequency values) to 0.83 for 12-word clusters. The decreases in recall as cluster size increases are primarily due to the fact that false negatives can be introduced in the approximation of burstiness. It is possible that the approximated average frequency $\bar{f}_{t,C}$ across time periods is greater than the actual occurrence over those time periods, resulting in an incorrectly low burstiness value. In this case, a cluster might be incorrectly discarded, which is reflected in reduced recall values.

Figure 3 shows our precision results, which exhibit behavior similar to the recall graph, in that precision also decreases with increasing cluster size. As is the case with recall, the burstiness approximation can also diminish precision. It is possible that the expected frequency $f_{t,C}$ of a cluster is greater than the actual occurrence frequency, resulting in a higher burstiness value than is actually the case. Here, a cluster might be incorrectly included.

We note that the precision curve has a noticeably sharper decline than the recall curve, from 1.0 for 2-word clusters (for which we have actual frequency values) to 0.42 for 12-word clusters. The cause of this is the frequency approximation in the first step. We get exact values for 2-word clusters, as our method computes and accumulates the frequencies of every 2-word cooccurrence for every time period. Thus, when we compute the candidate (frequent) 2-word clusters, we get every possible candidate,

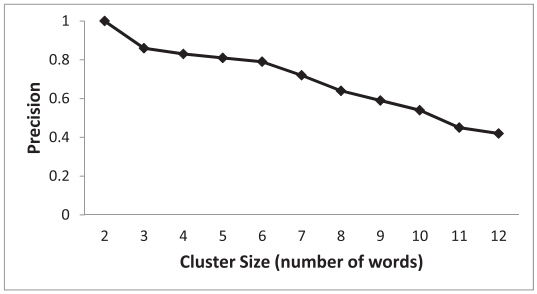

Fig. 3. Precision.

and no false positives. From that point on, as we generate larger clusters, we no longer keep count—larger clusters are generated by merging smaller candidate clusters. This, of course, is an approximation—if *AB*, *BC*, and *AC* are all frequent 2- word clusters, it is very likely, but not necessarily so that *ABC* will be a candidate frequent cluster as well. This approximation has the effect of introducing false positives in the system, which compound as the cluster size increases. This manifests itself in the more-sharply decreasing slope of the precision graph.

7.3. Scalability

We consider scalability across two dimensions: (a) post arrival rate and (b) query loads. We are interested in three distinct metrics. First, we measure how long it takes, on average, for the system to process a post, i.e., the time elapsed from the point when a post arrives at the system to when it has been fully processed by the Input Text Handler. We refer to this as the *Average Time in System* (ATIS). Second, we measure the *Throughput* of the system, i.e., how many posts the system can handle per second. Finally, we measure how long it takes, on average, for the system to process a query. We refer to this as the *Average Query Response Time* (AQRT). In this article, we report the measurement of ATIS and Throughput for varying post arrival rates and the measurement of AQRT for different query arrival rates.

We compare our method's performance for ATIS and Throughput with that of the approach described by Cataldi et al. [2010] to demonstrate the scalability of our method. The Cataldi approach does not provide for queries for trending topics within any given time interval, as provided by ours. Since there is no query processing aspect in Cataldi approach, we cannot compare AQRT.

We implemented the Cataldi algorithms in the experimental environment described in Section 7.1 and ran all ATIS and Throughput comparison experiments for both the *TrendMiner* and the Cataldi implementations. We note that the Cataldi approach requires a few parameters: (a) a *time slot*, which is comparable to our update interval; (b) a *number of slots to consider*, which is comparable to our query time frame, where all queries are current queries; and (c) a frequency support measure similar to our own. Since the parameters are comparable to our own, we use the same values as input to our system and the Cataldi module for each experimental run.

7.3.1. Average Time in System. The ATIS results are shown in Figure 4. In these experiments, we use a baseline query arrival rate of 10 queries/second. Here, we can see that the ATIS values are relatively stable up to 300 posts/second for the Cataldi approach, and up to 400 posts/second for our approach (labeled Online), indicating base system capacities. Up to these rates, arriving posts do not have to wait—they are processed immediately. Above these arrival rates, posts must wait in queues

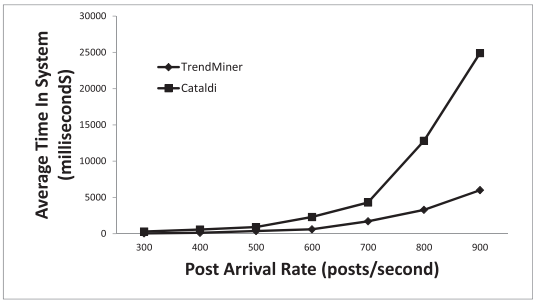

Fig. 4. Average time in system.

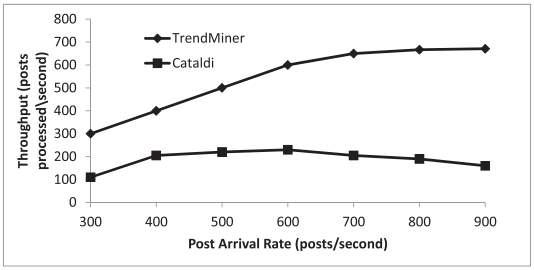

Fig. 5. Throughput.

for processing. This causes the ATIS to start to increase in both cases, with ATIS values reaching approximately 3.3 seconds for arrival rates of 900 posts/sec for the *TrendMiner* case. In contrast, for the Cataldi approach, the ATIS value is 24.9 (7.5 times the ATIS of our approach) at the same rate. The overhead of computing the correlation across all word pairs is the key bottleneck in the Cataldi approach—this bottleneck is the primary cause of the rapid increase in ATIS.

7.3.2. Throughput. The Throughput results are shown in Figure 5. In these experiments, we use a baseline post arrival rate of 10 posts/second. These results are in concordance with those shown in the ATIS graph. For our approach, Throughput continues to increase until the rate of incoming posts reaches 600 posts/second, after which Throughput begins to flatten out, indicating that the system has reached its processing capacity. In comparison, for the Cataldi approach, the throughput flattens out at the much lower incoming post rate of roughly 400 posts/second. This suggests that our system capacity is around 600 posts/second, and the Cataldi capacity is approximately 400 posts/sec in the experimental testbed. From this, we can conclude that the Cataldi approach reaches its bottleneck point roughly 200 posts/second earlier than the *TrendMiner* approach. This further demonstrates the suitability of our approach in online scenarios, as compared to the Cataldi approach.

7.3.3. Average Query Response Time. We measure the scalability of our system in terms of AQRT performance as the query arrival rate increases. Figure 6 shows how response time varies as the query arrival rate increases from 5 queries/second to 30 queries/second. We show three curves here, for post arrival rates of 200, 400, and 600 posts/second. The curves all show the same trend of increasing rate of slope increase as the query arrival rate increases. As we would expect, the response times are higher for the 400 and 600 posts/second curves than for the 200 posts/second

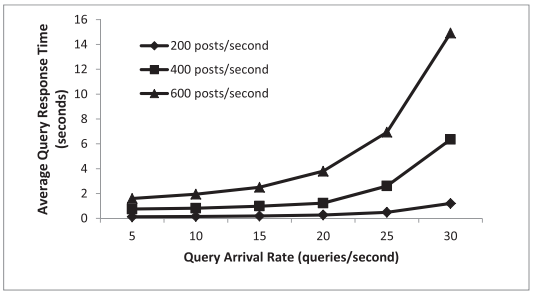

Fig. 6. Average query response time.

curve. Even at a high post arrival rate of 600 posts/second, a high query arrival rate of 30 queries/second, and all modules running on a single server, our system is still able to respond to queries within 15 seconds on average—which is within an order of magnitude of online responsiveness. With commercial-grade equipment, we would expect to achieve online or near-online responsiveness, even at these higher loads.

Finally, based on our ATIS and AQRT results, we consider how close to trending topic detection we have been able to get. In order for a topic to appear as trending in our system, there must be sufficient related posts in the update period to cross the frequency support threshold. When the last post required to push a topic across the threshold arrives, it must pass through the Input Text Handler, after which there is sufficient information in the HashTable to detect the trending topic. As shown in Figure 4, the ATIS at a post arrival rate of 600 posts/second is approximately 0.6 seconds. If a "current" query for trending topics were to arrive immediately after that final threshold-crossing post arrived, our system would process the request and return the corresponding trending topic in the results. As shown in Figure 6, at a query arrival rate of 10 queries/second, the AQRT is 1.9 seconds (using the 600 posts/second curve). If we sum these two values, we get a trending topic-detection delay of only 2.5 seconds, which is very close to online responsiveness [Broadwell 2004].

8. CONCLUSION

This article addresses the problem of finding trending topics in microblog posts with online responsiveness. We describe a method for handling high-volume post rates, and describe methods for concurrently finding trending topics. We compared the output of *TrendMiner* to trending topics from Twitter, and found that *TrendMiner* provided significantly higher contextual content than Twitter trending topics. In order to provide Internet-scale scalability and performance, our methods are approximate. Thus, we compared our results with those of an exhaustive clustering method in order to quantify accuracy. Our methods provide strong recall, which indicates that they are capable of identifying most trending clusters. Precision for our results is also high for cluster sizes in the range of 2–6 words, but accumulating errors for larger cluster sizes results in lower precision at higher cluster sizes. A comparison of our system with the Cataldi approach [Cataldi et al. 2010] suggests that our method scales 150% better and is capable of handling a much higher Twitter post arrival rate. We also measured the query response time performance of our system; these results suggest that our approach is acceptable for online applications.

This work has several limitations, which we plan to address in the future. First, the current approach does not consider the order of the words in trending topics. However, presenting results in an appropriate wording order would definitely result in an easier

to use interpretation of the resulting trending topics. Second, when the cluster size increases, the precision of the *TrendMiner* method decreases. In our future work we intend to improve the presentation by using appropriate word ordering and improve precision by refining our approximation strategy.

ACKNOWLEDGMENTS

We would like to gratefully acknowledge Abdur Chowdhury, Chief Scientist at Twitter, Inc., for his helpful comments and input.

REFERENCES

Agrawal, R. and Srikant, R. 1994. Fast algorithms for mining association rules. In *Proceedings of the 20th Very Large Database Conference.* 487–499.

Asur, S. and Huberman, B. A. 2010. Predicting the future with social media. In *Proceedings of the IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT)*. 492–499.

Benhardus, J. 2010. Streaming trend detection in Twitter. *UCCS REU For Artificial Intelligence, Natural Language Processing And Information Retrieval Final Report*, 1–7.

Bollen, J., Mao, H., and Pepe, A. 2011. Modeling public mood and emotion: Twitter sentiment and socio-economic phenomena. In *Proceedings of the International Conference on Weblogs and Social Media.*

Broadwell, P. M. 2004. Response time as a performability metric for online services. Tech. rep. UCB/CSD-04-1324, EECS Department, University of California, Berkeley.

Cataldi, M., Di Caro, L., and Schifanella, C. 2010. Emerging topic detection on Twitter based on temporal and social terms evaluation. *In Proceedings of the 10th International Workshop on Multimedia Data Mining*. 1–10.

Chang, J. H. and Lee, W. S. 2003. Finding recent frequent itemsets adaptively over online data streams. In *Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD).* 487–492.

Chang, J. H. and Lee, W. S. 2004. A sliding window method for finding recently frequent itemsets over online data streams. *J. Inform. Sci. Eng. 20*, 4, 753–762.

Chi, Y., Wang, H., Yu, P. S., and Muntz, R. R. 2004. Moment: Maintaining closed frequent itemsets over a stream sliding window. In *Proceedings of the IEEE International Conference on Data Mining (CDM)*. 59–66.

Forrester. 2012. Forrester ecommerce study: The two-second rule is critical. http://colderice.com/forrester-ecommerce-study-the-2-second-rule-is-critical/.

Giannella, C., Han, J., Pei, J., Yan, X., and Yu, P. S. 2004. Mining frequent patterns in data stream at multiple time granularities. In *Next Generation Data Mining,* H. Kargupta, A. Joshi, K. Sivakumar, and Y. Yesha Eds., AAAI/MIT, 191–212.

Glance, N., Hurst, M., and Tomokiyo, T. 2004. BlogPulse: Automated trend discovery for weblogs. In *Proceedings of the WWW Workshop on the Weblogging Ecosystem: Aggregation, Analysis and Dynamics.*

Hotho, A., Jaschke, R., Schmitz, C., and Stumme, G. 2006. Trend detection in folksonomies. *Semantic Multimedia,* 56–70.

Johnson, S. 2009. How Twitter will change the way we live. http://www.time.com/time/magazine/article/0,9171,1902818,00.html.

Kannan, A., Patzer, J., and Avital, B. 2010. Trendtracker: Trending topics on Twitter. http://vis.berkeley.edu/courses/cs294-10-sp10/wiki/images/d/d4/FinalPaper.pdf.

Khader, P., Scherag, A., Streb, J., and Roumlsler, F. 2003. Differences between noun and verb processing in a minimal phrase context: A semantic priming study using event-related brain potentials. *Cog. Brain Res. 17*, 2, 293–313.

Kwak, H., Lee, C., Park, H., and Moon, S. 2010. What is twitter, a social network or a news media? In *Proceedings of the 19th International Conference on the World Wide Web*. 591–600.

Li, H.-F. and Lee, S.-Y. 2009. Mining frequent itemsets over data streams using efficient window sliding techniques. *Expert Syst. Appl. 36*, 2, 1466–1477.

Li, H.-F., Ho, C.-C., and Lee, S.-Y. 2009. Incremental updates of closed frequent itemsets over continuous data streams. *Expert Syst. Appl. 36*, 2, 2451–2458.

Liang, X., Chen, W., and Bu, J. 2010. Bursty feature based topic detection and summarization. In *Proceedings of the 2nd International Conference on Computer Engineering and Technology.*

Manku, G. S. and Motwani, R. 2002. Approximate frequency counts over data streams. In *Proceedings of the 28th Very Large Data Base Conference (VLDB)*. 346–357.

Mathioudakis, M. and Koudas, N. 2010. TwitterMonitor: Trend detection over the Twitter stream. In *Proceedings of the International Conference on Management of Data*. 1155–1158.

Popescu, A.-M. and Pennacchiott, M. 2011. Dancing with the stars, NBA games, politics: An exploration of Twitter users' response to events. In *Proceedings of the 5th International AAAI Conference on Weblogs and Social Media*. 594–597.

Rui, H. and Whinston, A. 2012. Designing a social-broadcasting-based business intelligence system. ACM *Trans. Manage. Inf. Syst. 2,* 4, 1–19.

Twitter. Twitter posts. http://blog.twitter.com/2011/03/numbers.html.

Twitter, Inc. 2011. Year in review: Tweets per second. http://yearinreview.twitter.com/en/tps.html.

Twitter, Inc. 2012. Twitter turns six. http://blog.twitter.com/2012/03/twitter-turns-six.html.

Zhu, Y. and Shasha, D. 2002. Statstream: Statistical monitoring of thousands of data streams in real time. In *Proceedings of the 28th Very Large Data Base Conference*. 358–369.

Received April 2012; revised October 2012, November 2012; accepted November 2012