Parallel Clustering of High-Dimensional Social Media Data Streams

Xiaoming Gao

School of Informatics and Computing

Indiana University

Bloomington, IN, USA

gao4@umail.iu.edu

Judy Qiu

School of Informatics and Computing

Indiana University

Bloomington, IN, USA

xqiu@umail.iu.edu

*Abstract*—Social media data stream clustering has gained the attention of the research community in recent years. Of principle notice from latest work is that high-quality clusters can be generated by representing the data points with multiple high-dimensional vectors that reflect both textual content and social network information. Furthermore, similarity between data points are measured based on combinations of these vectors. However, due to the high cost of similarity computation, sequential implementations of even single-pass streaming algorithms are still not fast enough to match the speed of real-world social media data streams such as the ones generated by the Twitter streaming APIs. To bridge this gap, parallelization is necessary. This paper presents our efforts in parallelizing one state-of-the-art social media data stream clustering algorithm using the Storm stream processing engine. Specifically, we reveal and address two systems-level challenges. Firstly, most stream processing engines organize the distributed processing workers in the form of a directed acyclic graph (DAG), which makes it difficult to dynamically synchronize the state of parallel clustering workers. We tackle this challenge by creating a separate synchronization channel using a pub-sub messaging system. Secondly and more importantly, due to the sparsity of the high-dimensional vectors, the size of the cluster centroids grows quickly as new data points are assigned to the clusters. As a result, traditional synchronization strategies that directly broadcast the cluster centroids will eventually become very expensive and severely impact the scalability of the parallel algorithm. To solve this problem, we propose a new synchronization strategy that broadcasts the dynamic changes (i.e. the “deltas”) of the clusters rather than the whole vectors of the centroids. Evaluations using real datasets collected through the Twitter streaming API verify the correctness and scalability of our parallel algorithm, and demonstrate that it can eventually catch up with the speed of the Twitter gardenhose stream using less than 100 parallel clustering workers.

Keywords—social media data stream clustering; parallel algorithms; stream processing engines; high-dimensional data; synchronization strategies

# Introduction

As a fundamental data mining technique, clustering is valuable in many applications involving social media data stream analysis, such as meme identification [14][27], event detection [10], and social bots detection [14]. This is neatly demonstrated in Fig. 1, which illustrates the architecture of the DESPIC (Detecting Early Signatures of Persuasion in Information Cascades) platform [14] that is being developed by the Center for Complex Networks and Systems Research at Indiana University. This platform first applies a clustering component to group social messages into memes, and then uses a classification mechanism to identify different memes as content generated by human users or by social bots [15] for the purpose of persuasion or astroturfing.



1. DESPIC architecture for meme clustering and classification [14]

 Social media data streams come in the form of continuous sequences of separate social messages, e.g. Twitter tweets or Facebook status updates. The target of the clustering process is to group messages that carry similar social meaning together, while capturing the dynamic evolution of the streams that are closely related to social activities in the real world. For example, two tweets, “Step up time Ram Nation. #rowdyrams” and “Lovin @SpikeLee supporting the VCU Rams!! #havoc”, should be grouped into the same cluster because they both talk about VCU (Virginia Commonwealth University) basketball. Furthermore, the appearance of “@SpikeLee” in the cluster is an indicator of the event that the famous director Spike Lee was wearing a VCU T-shirt when watching the game between VCU and UMass at the side of the court on Mar 16th, 2013.

 When it comes to designing proper clustering algorithms, social messages demonstrate some unique characteristics that must be considered. For instance, the length of the textual content of a social message is normally short, which may make clustering methods purely based on lexical analysis ineffective [10][18][27]. Social messages also carry rich information about the underlying social network (e.g. through the functionality of ‘retweet’ and ‘mention’ on Twitter), which can be valuable for measuring the similarities among data points and clusters. In addition they may contain other metadata such as temporal/geographical information, hashtags, URLs, etc., which can also be leveraged to effectively guide the clustering process. Fig. 2 illustrates an example tweet received from the Twitter Streaming API [35]. Besides the textual content, hashtags, and URLs, it also contains information about the creation time and geolocation of the tweet, the author of the tweet, the user(s) mentioned in the tweet, and possible retweet relationship between tweets.



1. An example social message from Twitter Streaming API

Domain researchers in the area of social media data analysis have invested a great deal of effort in the last several years towards learning proper data representations and similarity metrics for generating high-quality clusters [10][14][18][27]. A common conclusion is that the data representation should not only describe the textual features of the social messages, but also capture the temporal, geographical, and social network information attached. In particular, [10] uses two high-dimensional vectors to describe each social message: one content vector that contains the textual word frequencies, and another binary vector housing the IDs of the social message’s recipients (e.g. the followers of a tweet’s author on Twitter). To compute the similarity between two social messages, a separate similarity is first computed using each vector, and then a linear combination of the two similarities is taken as the overall similarity between the two messages. It has been demonstrated that the quality of the result clusters can be significantly improved by using the combined similarity rather than just the textual content similarity. In [27], JafariAsbagh et al. propose to first group the social messages into ‘protomemes’ according to shared metadata such as hashtags and URLs, and then use the protomemes as input data points to the clustering algorithm. They use four high-dimensional vectors to describe each protomeme, and define a new ‘diffusion network’ vector to replace the full followers vector used in [10], which is hardly available in a practical streaming scenario. It is demonstrated that a combination of these new techniques can help generate better clustering results than previous methods when measured against a common ground truth data set.

To achieve efficient processing of the social media data streams, these special data representations and similarity metrics are normally applied in a single-pass clustering algorithm such as variants of online K-Means [2][10][27]. The algorithm can be further equipped with mechanisms like sliding time window [1][27], weighted data points [2][8][9][11], and outlier detection [8][10][16][27] to deal with the dynamic evolution of the streams. However, due to the high cost of similarity computation, sequential implementations of such single-pass streaming algorithms are not fast enough to match the speed of real-world social media data streams. For example, the fastest implementation presented in [10] can only process less than 20,000 tweets per hour, while the Twitter gardenhose stream [23] can generate more than 1,000,000 tweets in an hour. According to a test we carried out, it takes as many as 43.4 hours for a sequential implementation of the algorithm in [27] to process one hour’s worth of data collected through the gardenhose Twitter streaming API. It is clear that in order to catch up with the speed of real-world data streams, parallelization of such algorithms is necessary.

This paper presents our experience in parallelizing one state-of-the-art social media data stream clustering algorithm presented in [27], which is a variant of online K-Means with sliding time window and outlier detection mechanisms. Specifically, we use the Apache Storm [6] stream processing engine for data transmission and work load distribution, and demonstrate two systems-level challenges emerging from parallelization of such algorithms.

The first challenge concerns the fact that most stream processing engines organize the distributed processing workers in the form of a directed acyclic graph (DAG), which makes it difficult to dynamically synchronize the state of the parallel clustering workers without breaking the “live” processing of the stream. This is because the synchronization step requires the parallel workers to send their local updates either to each other or to a global updates collector, which will then broadcast the updated global state back to the parallel workers. Both ways will inevitably create cycles in the communication channel, which is not allowed in the DAG-oriented stream processing frameworks. To address this challenge, we create a separate synchronization channel by using the pub-sub messaging system ActiveMQ [5], and combine its functionality together with Storm to coordinate the synchronization process.

The second issue is that the sparsity of high-dimensional vectors may cause the cluster centroids to greatly increase in size with the influx of new data points to the clusters. As an example, consider Fig. 3, which illustrates a cluster containing the two tweets about VCU basketball as mentioned earlier. Because of the sparsity of the content vector (suppose the hashtags and user mentions are extracted as another separate vector) of each data point, they only overlap along one dimension “ram”. As a result, the length of the content vector of the centroid, which is computed as the average of the two data points, is close in sum to the length of the two separate vectors. Due to the high dimensionality of the vectors, this trend can continue as more data points are added, and the size of the centroid vectors can potentially grow very large. A sliding time window mechanism may help limit the total size by removing old data points from the clusters, but the whole centroids data would still become cumbersome to transfer over the network. Consequently, traditional synchronization strategies that directly broadcast the cluster centroids will eventually become infeasible and severely impact the scalability of the parallel algorithm. To solve this problem, we propose a new synchronization strategy that broadcasts the dynamic changes (i.e. the “deltas”) of the clusters rather than the whole vectors of the centroids. Since the size of the delta is small and stable, we are able to keep the synchronization cost at a low level and thus achieve much better scalability for the parallel algorithm. For the sake of simplicity, we name the traditional synchronization strategy *full-centroids strategy*, and our new synchronization strategy *cluster-delta strategy*.



1. An example of growing vector size of centroids

We use a real data set collected through the Twitter streaming API [35] to verify the effectiveness of our solutions and evaluate the scalability of our parallel algorithm. The results demonstrate that we can eventually catch up with the speed of the Twitter gardenhose stream using less than 100 parallel clustering workers. Although we target one specific algorithm in this paper, our new synchronization strategy is generally useful for the parallelization of many other algorithms using similar data representations and similarity metrics.

The rest of this paper is structured as follows. Section II discusses related work and their connections to our research. Section III gives a brief description of the sequential algorithm that we parallelize in this paper. Section IV explains our parallel algorithm implementation using Storm. Section V evaluates the effectiveness of our synchronization strategy and the scalability of our parallel algorithm. Section VI concludes and proposes potential future work.

# Related work

Data stream clustering algorithms have been an active research area for many years, and a condensed review of existing algorithms is available in [29]. Specifically for the problem of high-dimensional data stream clustering, techniques such as projected/subspace clustering [8][9][38] and density-based approaches [1][16][38] have been proposed and investigated. However, due to the unique data representations (multiple high-dimensional vectors from totally independent spaces) and similarity metrics used for social media data streams, it seems hard to apply these existing techniques to the case of social media streams. Therefore, our work in this paper inherits the high-dimensional data representation and similarity metrics that have been proven effective, and focuses on improving the efficiency of the clustering algorithm through parallel computing.

The algorithm in [10] uses sketch tables [12] to deal with the growing size of tweet followers network information maintained for the clusters. However, sketch tables only provide approximates of the real vector values and thus may impact the accuracy of the clustering results. In the case of our algorithm, since the size of the centroid vectors is constrained by the size of the sliding time window, we are not forced to use sketch tables in the cost of accuracy so far. For faster data streams or longer time windows, a sketch table-based implementation may eventually become more efficient in terms of both space and time for computing the similarities between data points and cluster centroids. Nonetheless, our cluster-delta synchronization strategy may still be more efficient than broadcasting the whole sketch tables in such cases because the sketch tables have to be large enough to ensure accuracy.

The most similar work to ours so far is the parallel implementation of the Sequential Leader Clustering [20] algorithm presented in [17], which also leverages Storm [6] for parallel processing and data stream distribution. Their parallel clustering algorithm is simplified, because it only considers the textual content of social messages and uses Locality-Sensitive Hashing [4] to guide the stream distribution, which helps eliminate the necessity for synchronization among the parallel clustering workers. However, such algorithms are not able to make use of the valuable social network information contained in the data streams. [25] proposes a distributed data stream clustering protocol based on sequential (a, b)-approximation algorithms for the K-Means problem, and presents theoretical study about the accuracy and efficiency of the protocol. However, it does not pay special attention to high-dimensional data, and only considers the situation within a single time window. Therefore, issues related to synchronization are not covered.

Compared with streaming databases such as Aurora [26] and Borealis [13], the functionality of our clustering workers in Storm is more complicated than the standard operators for evaluating SQL queries over data streams. In principle, it is also possible to implement our parallel algorithm using other stream processing engines such as Apache S4 [24] and Spark Streaming [28]. We choose Storm because its pull-based data transmission mode makes it easy to carry out controlled experiments at different parallelism levels. Compared with Spark Streaming, Storm’s programming model also allows more flexibility for us to implement and test different synchronization strategies. Interested readers may refer to [30] for a nice survey of major distributed stream processing frameworks.

# Sequential Clustering Algorithm

The sequential algorithm we parallelize was originally proposed in [27] for clustering memes in the Twitter streams of tweets. In order to generate high-quality clusters, the algorithm first groups tweets into ‘protomemes’, and then uses these protomemes as input data points to the clustering process. Therefore, we start this section by introducing the definition of a protomeme and its data representation.

## Protomemes and Clusters

A *protomeme* is defined as a set of tweets grouped together according to a shared entity of one of the following types:

* **Hashtags**. Tweets containing the same hashtag.
* **Mentions**. Tweets mentioning the same user. A mention is identified by a user’s screen name preceded by the ‘@’ symbol in the text body of a tweet.
* **URLs**. Tweets containing the same URL.
* **Phrases**. Tweets sharing the same phrase. A phrase is defined as the textual content of a tweet that remains after removing the hashtags, mentions, URLs, and after stopping and stemming [21].

We call these four types of entities *markers* of protomemes. Note that according to this definition, a tweet may belong to multiple protomemes. Each protomeme is represented by its marker and four high-dimensional vectors:

1. A binary *tid vector* containing the IDs of all the tweets in this protomeme: VT = [tid1, tid2, …, tidT];
2. A binary *uid vector* containing the IDs of all the users who authored the tweets in this protomeme: VU = [uid1, uid2, …, uidU];
3. A *content vector* containing the combined textual word frequencies for all the tweets in this protomeme: VC = [w1:f1, w2:f2, …, wC:fC];
4. A binary vector containing the IDs of all the users in the *diffusion network* of this protomeme. The diffusion network of a protomeme is defined as the **union** of the set tweet authors, the set of users mentioned by the tweets, and the set of users who have retweeted the tweets. We denote this *diffusion vector* as VD = [uid1, uid2, …, uidD].

A *cluster* is defined as a set of protomemes grouped together according to a certain similarity metric. Since a tweet may belong to multiple protomemes, clusters can have overlap with respect to tweets. The centroid of each cluster is also represented by four high-dimensional vectors, which are the averages of the corresponding vectors of all the protomemes in the cluster. We denote the vectors of the cluster centroid as V*T*, V*U*, V*C*, and V*D*. Being averages, V*T*, V*U*, and V*D* are no longer binary vectors; they contain the averaged frequencies for the corresponding user IDs.

To compute the *similarity* between a protomeme and a cluster, the **cosine similarity** between each vector of the protomeme and the corresponding vector of the cluster centroid is first computed. Then the **maximum value** of all these cosine similarities is taken as the overall similarity between the two. It has been demonstrated in [14] that for the purpose of generating high-quality clusters, taking the maximum is as effective as using an optimal linear combination of all the cosine similarities. There are multiple ways to define *distance* based on the similarity; the simplest form is *1 – similarity*.

## Sequential Clustering Algorithm

Fig. 4 illustrates the sketch of the sequential clustering algorithm from [27]. The algorithm controls its progress through a sliding time window that moves step by step. The length of a time step in seconds and the length of the time window in steps are given as input parameters. These are defined with regards to the timestamps of the social messages (i.e. tweets), not the natural time measured on the physical machines where the algorithm is running. Every time the sliding window advances, old protomemes falling out of the current window are deleted from the clusters and new ones are generated using the tweets from the latest time step. For every new protomeme, the algorithm first checks whether protomemes with the same marker have been previously assigned to a cluster. If so, the new protomeme will be directly added to the same cluster. Otherwise the algorithm will compute its similarity against all the clusters, and decide whether it is an outlier. If not, the protomeme is assigned to the most similar cluster. Otherwise, a new cluster is created and initialized with this new protomeme, then inserted into the list of all clusters by replacing either an empty cluster or the least recently updated one. In order to detect outliers, the algorithm maintains the mean *μ* and standard deviation *σ* of the similarities between all processed protomemes and the centroid of the clusters that they belong to. If the similarity between a new protomeme and its closest cluster is smaller than the mean by more than *n* standard deviations, then the protomeme is identified as an outlier. *μ* and *σ* are incrementally maintained in the same way as in [10].

**Algorithm** *TweetStreamClustering*

**Input parameters**:

 *K*: number of clusters;

 *t*: length of time step by which the time window advances;

 *l*: length of the time window in steps;

 *n*: number of standard deviations from the mean to identify

 outliers;

**begin**

Initialize global list of clusters *cl* as empty;

 Initialize global list of protomemes *gpl* as empty;

 Initialize the time window *tw* as empty;

 Initialize *μ*, *σ* to 0;

 **while** **not** end of stream **do**

advance the time window *tw* by *t*;

**let** *npl* = list of protomemes generated from the tweets in *t*;

**if** *cl* is empty **then**

initialize *cl* using *K* random protomemes in *npl*;

 remove these *K* protomemes from *npl*;

**endif**

**for** each protomeme *p* in *gpl* that is older than the current *tw*

 delete *p* from *gpl* and the cluster it belongs to;

**endfor**

**for** each new protomeme *p* in *npl*

 **if** *p.marker* has been previously assigned to a cluster *c* **then**

add *p* to *c*and update the centroid of *c*;

 **else**

 **let** *c* = the cluster in *cl* whose centroid is most similar to *p;*

 **if** *sim(p, c)* > *μ – n \* σ* **then**

add *p* to *c*and update the centroid of *c*;

 **else**

 create a new cluster *c’* containing only one protomeme *p*;

 **if** there is an empty cluster in *cl* **then**

 replace the empty cluster with *c’*;

 **else**

 replace the least recently updated cluster in *cl* with *c’*;

 **endif**

 **endif**

 **endif**

add *p* to *gpl*;

dynamically maintain*μ* and *σ*;

**endfor**

 **endwhile**

**end**

1. The social media stream clustering algorithm from [27]

The quality of clusters generated by this algorithm was evaluated in [27] using a ground truth dataset collected from the Twitter gardenhose stream [23] during a week in 2013, which includes all the tweets containing the Twitter trending hashtags [36] identified during that time. Specifically, a variant of the *Normalized Mutual Information* (NMI) [22] measurement, LFK-NMI [3], which is specially designed for the case of overlapping clusters, was computed between the result clusters of the algorithm and the ground truth clusters that are identified beforehand. The results in [27] show that this algorithm can achieve higher LFK-NMI values than previous state-of-the-art methods, including the one presented in [10]. We use the same ground truth dataset and LFK-NMI measurement to verify the effectiveness of our parallel implementation of the algorithm in section V.

## Opportunities and Difficulties for Parallelization

To investigate the possible opportunities and difficulties for the parallelization of the algorithm, we run the sequential algorithm on a raw dataset (without any filtering) containing six minutes of tweets (2014-08-29 05:00:00 to 05:05:59) collected from the Twitter gardenhose stream. We fix the parameters *K*, *l*, and *n* to 120, 6, and 2, and vary the length of a time step to collect some important runtime statistics that are informative to the development of the parallel version of the algorithm.

1. Runtime Statistics For the Sequential Algorithm

| Time Step Length (s) | Total Length of Content Vector | Similarity Compute time (s) | Centroids Update Time (s) |
| --- | --- | --- | --- |
| 10 | 47749 | 33.305 | 0.068 |
| 20 | 76146 | 78.778 | 0.113 |
| 30 | 128521 | 209.013 | 0.213 |

Table I presents the numbers collected for the last time step of the whole clustering process when the time step length is increased from 10 seconds to 30 seconds (which means the time window length is increased from 60 to 180 seconds). The numbers for the other time steps follow a similar pattern. The second column measures the total length of the content vectors of all the cluster centroids at the end of the last time step; the third column measures the time spent on computing the similarities between protomemes and cluster centroids in that time step; and the fourth column measures the time spent on updating the vectors of the cluster centroids.

Some interesting observations are inspiring for our research into the parallelization of the algorithm:

First of all, the whole clustering process is dominated by the computation of similarities. The relative ratio of **similarity compute time / centroids update time** increases from 490 to 981 as the length of the time window is increased. This implies the feasibility of parallelizing the similarity computation, and processing the updates of centroids in a collective way. Furthermore, the longer the time window is, the more we can benefit from parallelization.

We also observed that the content vector size of the centroids expands as the length of the time window increases. In fact, the other vectors demonstrate the same trend. Therefore, the traditional synchronization strategy that broadcasts the whole centroids of all clusters will eventually become very time-consuming for lengthy time windows, and severely impact the scalability of the parallel algorithm. This is the reason why we designed the new cluster-delta synchronization strategy, which will be presented in section IV.

# Parallel Implementation on Storm

## Storm

Apache Storm is a stream processing engine designed to support large-scale distributed processing of data streams. It defines a stream as an unbounded sequence of *tuples*, and provides an easy-to-use event-driven programming model to upper level applications. Stream processing applications are defined in the form of *topologies* in Storm, as exemplified in Fig. 5. There are two types of *processing elements* in a topology, *spouts* and *bolts,* which are organized into a DAG through the streams connecting them. A spout is a source of streams that generates new tuples and injects them into the topology. A bolt can consume any number of input streams, do some processing to each tuple of the streams, and potentially generate and emit new tuples to the output streams. To define a topology, the application only needs to provide implementation logics of spouts and bolts, specify the runtime parallelism level of each type, and configure the data distribution patterns among them. The Storm framework will automatically take care of system management issues including data transmission, parallel spouts/bolts execution, work load distribution, and fault tolerance.



1. An example topology in Storm



1. Storm architecture

Fig. 6 illustrates the architecture of a Storm cluster. The whole cluster consists of two types of nodes: one master node and multiple worker nodes. The master node runs a daemon process called *Nimbus*, which is responsible for assigning spout and bolt tasks to the worker nodes and monitoring their status for failures. Every worker node runs a *Supervisor* daemon process, which manages the resources on the local node, and accepts task assignments from the *Nimbus*. The spout and bolt tasks are launched as parallel threads in *worker processes*. The number of worker processes on each node is configurable as a system parameter. The number of threads to run for each type of spout and bolt in a topology can also be configured through the parallelism parameters. The coordination between the *Nimbus* and the *Supervisors* is done by using Zookeepers [7].

Storm adopts the ‘pull-based’ message passing model between the processing elements. Bolts pull messages from the upstream bolts or spouts. This ensures that bolts will never get excessive amounts of workload that they cannot handle. Therefore, overflow can only happen at the spouts. This model offers us an easy way to carry out controlled experiments for testing our algorithm at different levels of parallelism. For example, we can implement spouts that generate streams by reading data from a file, and control their paces based on the number of acknowledgements received for tuples that have been processed. As a result, the whole topology will never get overwhelmed no matter how slowly the bolts are working.

By using a tuple anchoring mechanism, Storm can guarantee that each tuple coming off a spout is fully processed at least once [31]. More sophisticated semantics such as exactly-once-processing are also possible through proper configuration and implementation [32].

## Implementation with Cluster-Delta Synchronization Strategy

We implement the parallel version of the algorithm in a Storm topology, as illustrated in Fig. 7. There is one type of spout, *Protomeme Generator Spout*, and two types of bolts, *Clustering Bolt* and *Synchronization Coordinator* *Bolt*, in the topology. For simplicity, we call them *protomeme generator*, *cbolt*, and *sync coordinator*. During runtime, there is one instance of the protomeme generator, multiple instances of cbolts working in parallel, and one instance of sync coordinator. A separate synchronization channel is created between the cbolts and the sync coordinator by using the ActiveMQ pub-sub messaging system [5]. ActiveMQ allows its client applications to connect to *message brokers*, and register themselves as *publishers* or *subscribers* to various *topics*. Publishers can produce messages and publish them to a certain topic, and the message broker will automatically deliver the messages to all the subscribers of that topic. In our topology, the sync coordinator is registered as a publisher to a topic named “clusters.info.sync”, and all the cbolts are registered as subscribers to this topic. The lifetime of the whole topology can be divided into two phases, an *initialization phase* and a *running phase*. We explain the working mechanism of each type of spout and bolt in both phases.



1. Storm topology for the parallel stream clustering algorithm

***Protomeme Generation and Processing***

During the **initialization phase**, every processing element reads some information from a bootstrap file. Specifically, the protomeme generator reads the start time of the current time step, the length of a time step in seconds, and the length of a time window in steps. After reading such information, it can either connect to an external stream of tweets or open a file containing tweets to get ready for generating protomemes in the next time step. For the cbolts and sync coordinator, besides the same time window configuration parameters as mentioned above, they will also read the input parameter *n* (number of standard deviations for outlier detection), and a list of initial clusters. The initial clusters can be generated by running either a parallel batch clustering algorithm, or the sequential stream clustering algorithm over a small batch of data from recent history. After reading this information, the cbolts and sync coordinator will all have the same picture of the initial clusters. The initial values of *μ* and *σ* are then generated based on the protomemes contained in the initial clusters. Every cbolt will also generate a mapping between the protomeme markers and the clusters they belong to. After this, the cbolts and sync coordinator will connect to an ActiveMQ message broker and register themselves as subscribers and the publisher. Since the cbolt tasks are running as threads in worker processes, they will first go through an **election** step to select one **representative thread** within each process. Only the representative thread will be registered as a subscriber, and the synchronization message received will be shared among the threads in the same process. This election step can significantly reduce the amount of data transmission during the synchronization stage.

During the **running phase**, the protomeme generator keeps reading and buffering tweets for the ‘current’ time step, until it sees a tweet with a timestamp falling into the next time step. Then it generates protomemes using the buffered tweets. Every protomeme is associated with a *creation timestamp* and an *ending timestamp*, which are set respectively to the timestamp of the earliest and latest tweet in the protomeme. To facilitate the construction of diffusion vectors of protomemes, an **in-memory index structure** is maintained to record the mapping between each tweet ID and the set of user IDs who have retweeted it. To construct the diffusion vector of a protomeme, the user IDs of the tweet authors and the user IDs mentioned in its tweets are first added to the vector. Then the index is queried for each tweet ID of the protomeme, and the corresponding user IDs found in the index are also added to the vector. The protomeme generator emits one tuple to its output stream for every newly generated protomeme. The tuples are evenly distributed among all the parallel cbolts based on the hash values of their markers. Therefore, protomemes generated in different time steps but sharing the same marker will always be processed by the same cbolt.

At the cbolts and sync coordinator, protomemes are processed in small batches. A *batch* is defined as the number of protomemes to process, which is normally configured to be much smaller than the total number of protomemes generated in a single time step. Upon receiving a protomeme tuple, the cbolt first checks its creation timestamp to see if it starts a new time step. If so, the cbolt will first advance the current time window by one step, and delete all the old protomemes falling out of the time window from the clusters they belong to. Then it tries to do outlier detection and protomeme-cluster assignment in the same way as in the sequential algorithm, based on the current clusters and *μ*, *σ* values for the current batch. If the protomeme is detected as an outlier, an *OUTLIER* tuple containing the protomeme will be emitted to the sync coordinator. If the protomeme can be assigned to a cluster, a *PMADD* tuple will be emitted, which contains the protomeme, the ID of the target cluster, and the similarity between the two. Note that the cbolt does not immediately create a new cluster with the outlier protomeme, because outlier protomemes detected by different cbolts may be similar to each other and thus should be grouped into the same cluster. Such global grouping can only be done by the sync coordinator, which collects *OUTLIER* tuples generated by all the cbolts. For the case of *PMADD*, the centroid of the corresponding cluster is not immediately updated either. Instead, clusters are only updated during the synchronization between two consecutive batches. This ensures that within the same batch, different cbolts are always comparing their received protomemes against the same set of global clusters.

Within each batch, the sync coordinator maintains a list of ‘cluster delta’ data structures and another list of outlier clusters. Upon receiving a *PMADD* tuple, it will simply add the protomeme contained in the tuple to the delta structure of the corresponding cluster, and change the latest update time of the delta structure to the ending timestamp of the protomeme in case the ending timestamp is larger. Since the sync coordinator collects *PMADD* from all parallel cbolts, the delta structures will contain the global updates to each cluster. For an *OUTLIER* tuple, it will first check whether the protomeme contained in the tuple can be assigned to any existing outlier cluster. If so, it is simply added to that outlier cluster; otherwise a new outlier cluster is created and appended to the list of outlier clusters. The latest update time of the corresponding outlier cluster is also updated with the ending timestamp of the protomeme in case the ending timestamp is larger. After processing each tuple, the values of *μ* and *σ* are also dynamically updated.

***Synchronization***

In order to coordinate the synchronization process, the sync coordinator counts the number of *PMADD* and *OUTLIER* tuples received in the current batch. If the total number reaches the batch size, it will publish a *SYNCINIT* message to the ActiveMQ topic. Since the size of this message is very small, it can be delivered to the cbolts within milliseconds. Upon receiving this message, each cbolt will temporarily stop the processing of incoming protomemes, and emit a *SYNCREQ* tuple to its Storm output stream. After receiving *SYNCREQ* from all the cbolts (which means all of them are temporarily frozen and waiting for synchronization), the sync coordinator will first create delta structures for all the outlier clusters, and append them to the list of delta structures. The delta structure for an outlier cluster simply contains all the protomemes in the cluster, because they are all from the current batch. Then the whole list of delta structures is sorted by their latest update time, and the top *K* delta structures with the highest values are selected for constructing the *CDELTAS* message. If any one of the top *K* delta structures is created from an outlier cluster, it is specially marked and assigned a cluster ID of a deleted obsolete cluster. Besides these delta structures, the latest values of *μ* and *σ* are also added to the *CDELTAS* message, which is then published through ActiveMQ. Upon receiving the *CDELTAS* message, every cbolt will update their local copy of clusters and *μ*, *σ* values to a new globally consistent state, then resume processing the protomemes for the next batch. Note that the *SYNCINIT* step and the temporary stopping of the cbolts are necessary to ensure that protomemes processed by different cbolts and received by the sync coordinator are always handled with regards to the same global view of the clusters. Since the size of the *CDELTA* messages is normally small and stable, the synchronization step can usually finish in a short time, as will be demonstrated in section V.

## Implementation with Full-Centroids Synchronization Strategy

To verify the effectiveness of our cluster-delta synchronization strategy, we implement another version of the parallel algorithm using the full-centroids strategy for the purpose of comparison. The protomeme generation and processing logics of the full-centroids version are mostly the same as the cluster-delta version. Regarding the implementation details, there are some major differences caused by the full-centroids synchronization strategy:

During the processing time of each batch, the sync coordinator will maintain a full list of existing clusters, instead of the delta structures for them. During the synchronization time, instead of the *CDELTAS* message, it will generate a *CENTROIDS* message, which directly contains the whole centroid vectors of the clusters with the top *K* latest update times. Upon receiving the *CENTROIDS* message, every cbolt will directly use the centroid vectors contained in the message to replace the centroids of the old clusters.

Since the cbolt directly receives the centroid vectors instead of the incremental protomemes of each cluster, it can no longer maintain a full record of all the protomemes in the clusters. Therefore, the task of new time step detection and old protomeme deletion is moved to the sync coordinator. Since the centroids update time is negligible compared to the similarity compute time, this has little impact on the overall performance of the algorithm.

# Evaluation of the Parallel Algorithm

We verify the correctness of our parallel algorithm by comparing its results with the sequential implementation, and then evaluate its efficiency and scalability through comparison with the full-centroids synchronization strategy. Our evaluation tests are done on a private eight-node cluster that has a total of 128 CPU cores. The hardware configuration of each node is listed in Table II. Each node runs RHEL 6.5, Java 1.7.0\_45, and Apache Storm 0.9.2. Apache ActiveMQ 5.4.2 is deployed on the same node where the Strom Nimbus runs. Each node is configured to run at most four Storm worker processes, and the parallel instances of spouts and bolts are launched as threads spawned by these worker processes. The maximum heap size of each worker process is set to 11GB. Message compression with zip is enabled for ActiveMQ, and only one message broker is used in all tests of the parallel implementations.

1. Hardware Configuration of Each Node

| **CPU** | **RAM** | **Hard Disk** | **Network** |
| --- | --- | --- | --- |
| 4 \* 4 Quad-Core AMD Opteron 8356 2.3G Hz | 48GB | 4 TB HDD + 1TB SSD | 1Gb Ethernet |

## Correctness Verification

To test our algorithm’s correctness, we use the same ground truth dataset and LFK-NMI measurement as [27]. The LFK-NMI value between two sets of result clusters is a number between 0.0 and 1.0 that indicates the degree of matching between the two sets of clusters. A value of 1.0 corresponds to a perfect matching, while a value of 0.0 means that the two sets of clusters have nothing in common. The ground truth dataset was collected from the Twitter gardenhose stream [23] within the week of 2013-03-23 to 2013-03-29. It includes all the tweets containing the Twitter trending hashtags [36] identified during that time.

To carry out the correctness tests, we first define the ground truth clusters as the sets of tweets corresponding to each trending hashtag, i.e. all the tweets sharing a common trending hashtag are grouped into one separate cluster. Note that since a tweet may contain multiple trending hashtags, the ground truth clusters may have overlaps. We then remove the trending hashtags from the content of all the tweets, and run both the sequential implementation from [27] and our parallel implementation over the remaining dataset. As a result, protomemes corresponding to the trending hashtags will not be created and used as input data points to the clustering process. Finally, we compute three LFK-NMI values: one between the results of the sequential algorithm and the ground truth clusters, one between the results of the parallel algorithm and the ground truth clusters, and one between the results of the sequential and the parallel algorithm. We use the same input parameters as the experiments completed in [27]: *K* = 11, *t* = 60 minutes, *l* = 6, and *n* = 2. For the parallel algorithm, we use two parallel cbolts and a batch size of 40.

Table III presents the LFK-NMI values using the final clusters generated by the two algorithms. The high value of 0.728 in the first column indicates the clusters generated by our parallel implementation match very well with the results of the original sequential implementation in [27]. Moreover, values in the second and third column suggest that when measured against the same ground truth clusters, our parallel implementation can achieve a degree of matching comparable to the sequential implementation. These numbers together verify that our parallel implementation is correct and can generate results that are consistent with the sequential algorithm. The value 0.169 is consistent with the original test results in [27]. Furthermore, the slightly higher value of 0.185 indicates that processing the protomemes in small batches may be helpful for improving the quality of the clusters.

1. LFK-NMI Values for Correctness Verification

| **Parallel vs. Sequential** | **Sequential vs. ground truth** | **Parallel vs. ground truth** |
| --- | --- | --- |
| 0.728 | 0.169 | 0.185 |

## Performance Evaluation

To evaluate the performance and scalability of our parallel algorithm, we use a raw dataset collected from the Twitter gardenhose stream without applying any type of filtering. It contains a total number of 1,284,935 tweets generated within one hour (from 05:00:00 AM to 05:59:59 PM) on 2014-08-29. We first run the sequential algorithm over the whole dataset using input parameters *K* = 240, *t* = 30 seconds, *l* = 20, and *n* = 2, and measure the total processing time. Note that the time window has a length of 10 minutes and thus may contain a large number of protomemes. Then we run the two versions of parallel implementation at different levels of parallelism, and measure the corresponding total processing time, speedup, and other important statistics. We use the clusters generated for the first 10 minutes of data as the bootstrap clusters, and process the following 50 minutes of data using the parallel algorithms. The average number of protomemes generated in each time step is 19908, and the batch size is set to 6144 for the parallel algorithms.

The total processing time of the sequential algorithm is 156,340 seconds (43.43 hours), and the time spent on processing the last 50 minutes of data is 139,950 seconds (38.87 hours). The statistics for the two parallel implementations are given in Table IV and Table V respectively. The numbers in brackets in the first column tell how many Storm worker processes were used for hosting the cbolt threads. These correspond to the total numbers of ActiveMQ receivers in each run. Here we list the numbers that delivered the best overall performance. The length of the synchronization message in the last column is measured before ActiveMQ does any compression. Fig. 8 compares the scalability of the two versions of parallel implementations.

1. Statistics for Full-centroids Version

| **Number of cbolts (worker processes)** | **Total processing time (sec)** | **Compute time / sync time** | **Sync time per batch (sec)** | **Avg. length of sync message** |
| --- | --- | --- | --- | --- |
| 3 (1) | 67603 | 31.56 | 6.45 | 22113520 |
| 6 (1) | 35207 | 15.53 | 6.51 | 21595499 |
| 12 (2)  | 19228 | 7.79 | 6.60 | 22066473 |
| 24 (4) | 10970 | 3.95 | 6.76 | 22319413 |
| 48 (7) | 6818 | 1.92 | 7.09 | 21489950 |
| 96 (28) | 5804 | 0.97 | 8.77 | 21536799 |

1. Statistics for Cluster-delta Version

| **Number of cbolts (worker processes)** | **Total processing time (sec)** | **Compute time / sync time** | **Sync time per batch (sec)** | **Avg. length of sync message** |
| --- | --- | --- | --- | --- |
| 3 (1) | 50377 | 289.18 | 0.54 | 2525896 |
| 6 (1) | 22888 | 124.62 | 0.56 | 2529779 |
| 12 (2) | 11474 | 58.45 | 0.58 | 2532349 |
| 24 (4) | 6140 | 27.44 | 0.64 | 2544095 |
| 48 (7) | 3333 | 11.96 | 0.76 | 2559221 |
| 96 (28) | 1999 | 5.95 | 0.89 | 2590857 |



1. Scalability comparison between two versions of parallel implemantations

Table IV demonstrates that due to the large size of the cluster centroids, the full-centroids strategy generates a large synchronization message at the level of 20+ megabytes, and incurs a long synchronization time in every batch. Moreover, the synchronization time gets even longer as the number of parallel cbolts increases, because the single ActiveMQ broker needs to send the large message to more subscribers. In particular, the total processing time for the case of 96 parallel cbolts is dominated by synchronization. As a result, the corresponding algorithm demonstrates bad scalability in Fig. 8, and almost stops getting faster after the point of 48 parallel cbolts.

In comparison, the cluster-delta strategy generates a much smaller synchronization message, and thus keeps the per-batch synchronization time at a low level, as shown in Table V. The zip compression of ActiveMQ provides a compression ratio of about 1:6, so the actual message size sent over the network is less than 500KB. As the number of parallel cbolts increases, the computation time covers the major part of the total processing time for all cases. Therefore, the parallel implementation using the cluster-delta strategy can achieve a near-linear scalability for up to 48 parallel cbolts. Overall, it demonstrates a good sub-linear scalability. Note that even for the case of 96 parallel cbolts, the per-batch synchronization time is still relatively low. The major reason for the relatively low speedup of 70.0 is lack of computation, because each cbolt only processes about 64 protomemes in each batch. For the case of longer time steps or faster data rate, we can extend the near-linear-scalability zone to larger numbers of parallel cbolts by further increasing the batch size. Using 96 parallel cbolts, our algorithm finishes processing the 50 minutes’ worth of data in 1999 seconds (33.3 minutes), thus catching up with the speed of the Twitter gardenhose stream.

# Conlusions and Future Work

In summary, we have had success in our method of improving the efficiency of social media stream clustering algorithms by means of parallelization. Two important lessons are learnt from our experience:

Firstly, the distributed stream processing engines provide an easy way to develop and deploy large-scale stream processing applications. However, in order to properly coordinate the dynamic synchronization between parallel processing workers, their DAG-oriented processing models will need to be combined with facilitating tools such as pub-sub messaging systems. Whether such synchronization facilitating mechanisms should be directly built into the stream processing engines, as well as how this can be done, could become an interesting research issue for the distributed systems community.

Secondly, different parallelization and synchronization strategies should be designed and applied depending on the data representations and similarity metrics. For example, our experience shows that the high-dimensionality and sparsity of the data representation could cause problems for both computation and communication. By replacing the traditional full-centroids synchronization strategy with the new cluster-delta strategy, our parallel algorithm is able to achieve good sub-linear scalability, and eventually catches up with the speed of the real-world Twitter gardenhose stream with less than 100 parallel workers.

There are several interesting directions that we can further explore in the future. First of all, the scalability of our parallel algorithm can be further improved by using more advanced collective communication libraries such as Harp [33] for synchronization. Instead of using a “gather and broadcast” communication model, Harp can organize the parallel cbolts in the form of a communication chain, so that the cluster deltas generated by each cbolt instance can be transmitted through all the other instances in a pipeline fashion. According to our previous experience [37] in applying this technique in the Twister iterative MapReduce framework [19], it can significantly reduce the synchronization time and thus help the algorithm achieve near linear scalability. With this improved synchronization speed, we will try to approach the data rate of the Twitter firehose stream [34], which is about 10 times faster than gardenhose. Secondly, in order to support higher data speed and larger time window sizes, we will try to apply the sketch table technique as described in [12] in the clustering bolts and test its impact on the accuracy and efficiency of the whole parallel program. Finally, fluctuations in speed can be observed in many real-world data streams. Therefore, making the parallel algorithm more elastic and adaptive to the dynamic data speed changes will also be an interesting research issue.

##### Acknowledgment

We would like to thank the authors of [27] for their generous help in explaining their sequential algorithm to us.

##### References

1. A. Amini, T. Y. Wah. DENGRIS-Stream: a density-grid based clustering algorithm for evolving data streams over sliding window. In Proceedings of the 2012 International Conference on Data Mining and Computer Engineering (ICDMCE 2012).
2. A. King. Online K-Means clustering of nonstationary data. Course project report. 2012. Available at <http://coursepin.com/pin/3970>.
3. A. Lancichinetti, S. Fortunato, and J. Kertész. Detecting the overlapping and hierarchical community structure in complex networks. New Journal of Physics, 11(3):033015, 2009.
4. A. Z. Broder. On the resemblance and containment of documents. In Proceedings of the Compression and Complexity of Sequences 1997 (SEQUENCES 1997).
5. Apache ActiveMQ. <http://activemq.apache.org/>
6. Apache Storm. <https://storm.incubator.apache.org/>
7. Apache Zookeeper. <http://zookeeper.apache.org/>
8. C. C. Aggarwal, J. Han, J. Wang, P. S. Yu. A framework for projected clustering of high dimensional data streams. In Proceedings of the 30th international conference on very large data bases (VLDB 2004).
9. C. C. Aggarwal, J. Han, J. Wang, P. S. Yu. On high dimension projected clustering of uncertain data streams. Data Mining and Knowledge Discovery. 10(3):251–273
10. C. C. Aggarwal, K. Subbian. Event detection in social streams. In Proceedings of SIAM International Conference on Data Mining, 2012.
11. C. C. Aggarwal, P. S. Yu. A framework for clustering massive text and categorical data streams. Knowledge and Information Systems. 24(2): 171-196. August 2010.
12. C. C. Aggarwal. A framework for clustering massive-domain data streams. In Proceedings of IEEE 25th International Conference on Data Engineering (ICDE 2009).
13. D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J. Hwang, et al. The design of the Borealis stream processing engine. In Proceedings of the 2nd Biennial Conference on Innovative Data Systems Research (CIDR 2005).
14. E. Ferrara, M. JafariAsbagh, O. Varol, V. Qazvinian, F. Menczer, A. Flammini. Clustering memes in social media. In Proceedings of the 2013 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM 2013).
15. E. Ferrara, O. Varol, C. Davis, F. Menczer, A. Flammini. The rise of social bots. Unpublished and under review. Available at <http://arxiv.org/abs/1407.5225>.
16. F. Cao, M. Ester, W. Qian, A. Zhou. Density-based clustering over an evolving data stream with noise. In Proceedings of 2006 SIAM Conference on Data Mining.
17. G. Wu,O. Boydell, P. Cunningham. High-throughput, Web-scale data stream slustering. In Proceedings of the 4th Web Search Click Data workshop (WSCD 2014).
18. H. Becker, M. Naaman, L. Gravano. Learning similarity metrics for event identification in social media. In Proceedings of the 3rd ACM International Conference on Web Search and Data Mining (WSDM 2010).
19. J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S. Bae, J. Qiu, G. Fox. Twister: a runtime for iterative MapReduce. In Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC '10).
20. J. Hartigan. Clustering algorithms. John Wiley and Sons, New York, 1975.
21. J. Zobel, and A. Moffat. Inverted files for text search engines. ACM Computing Surveys, 38(2) - 6, 2006.
22. L. Danon, A. Díaz-Guilera, J. Duch, and A. Arenas. Comparing community structure identification. Journal of Statistical Mechanics: Theory and Experiment, 2005(09):P09008, 2005.
23. L. Gannes. Twitter adopts graded approach for developer access to tweets. 2010. News article available at http://allthingsd.com/20101110/twitter-firehose-too-intense-take-a-sip-from-the-garden-hose-or-sample-the-spritzer/
24. L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: distributed stream computing platform. In Proceedings of 2010 IEEE International Conference on Data Mining Workshops (ICDMW 2010).
25. M. Callau-Zori. INDICIa: a new distributed clustering protocol. In Proceedings of 28th ACM Symposium On Applied Computing (SAC13).
26. M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, Y. Xing, and S. Zdonik. Scalable distributed stream processing. In Proceedings of 1st Biennial Conference on Innovative Data Systems Research (CIDR 2003).
27. M. JafariAsbagh, E. Ferrara, O. Varol, F. Menczer, A. Flammini. Clustering memes in social media streams. Unpublished and under review.
28. M. Zaharia, T. Das, H. Li, S. Shenker, I. Stoica. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In Proceedings of the 4th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 2012).
29. S. Ding, F. Wu, J. Qian, H. Jia, F. Jin. Research on data stream clustering algorithms. Artificial Intelligence Review. January 2013.
30. S. Kamburugamuve. Survey of distributed stream processing for large stream sources. Technical report. 2013. Available at <http://grids.ucs.indiana.edu/ptliupages/publications/survey_stream_processing.pdf>.
31. Storm documentation - guaranteeing message processing. <https://storm.incubator.apache.org/documentation/Guaranteeing-message-processing.html>
32. Storm documentation - transactional topologies. <https://storm.incubator.apache.org/documentation/Transactional-topologies.html>
33. The Harp project. <https://github.com/jessezbj/harp-project>
34. Twitter firehose stream. <https://dev.twitter.com/streaming/firehose>
35. Twitter streaming API. <https://dev.twitter.com/streaming/overview>
36. Twitter trends. <https://mobile.twitter.com/trends>
37. X. Gao, J. Qiu. Social Media Data Analysis with IndexedHBase and Iterative MapReduce. In Proceedings of the 6th Workshop on Many-Task Computing on Clouds, Grids, and Supercomputers (MTAGS 2013).
38. Y. Sun and Y. Lu. A grid-based subspace clustering algorithm for high-dimensional data streams. Lecture Notes in Computer Science Volume 4256, 2006, pp 37-48.