Scalable Distributed Event Detection for Twitter

Richard McCreadie*, Craig Macdonald* and Iadh Ounis*
*School of Computing Science
University of Glasgow
Email: firstname.lastname@glasgow.ac.uk

Miles Osborne† and Sasa Petrovic†
†School of Informatics
University of Edinburgh
Email: miles@inf.ed.ac.uk, s.petrovic@sms.ed.ac.uk

***Abstract*—Social media streams, such as Twitter, have shown themselves to be useful sources of real-time information about what is happening in the world. Automatic detection and tracking of events identified in these streams have a variety of real-world applications, e.g. identifying and automatically reporting road accidents for emergency services. However, to be useful, events need to be identified within the stream with a very low latency. This is challenging due to the high volume of posts within these social streams. In this paper, we propose a novel event detection approach that can both effectively detect events within social streams like Twitter and can scale to thousands of posts every second. Through experimentation on a large Twitter dataset, we show that our approach can process the equivalent to the full Twitter Firehose stream, while maintaining event detection accuracy and outperforming an alternative distributed event detection system.**

***Keywords*-System analysis and design, Event detection, Distributed processing, Large-scale systems, Scalability**

I. Introduction

Real-time event detection involves the identification of newsworthy happenings (events) as they occur. These events can be mainstream, e.g. when a plane crashes into the Hudson river, or local events, e.g. a house fire nearby. Automatic online event detection systems use live document streams to detect events. For instance, streams of newswire articles from multiple newswire providers have previously been used for event detection [1], [2]. However, the role of the public in the online news space has changed, with social media websites such as Twitter now being used to report events as they happen [4], sometimes from the scene using mobile devices. Intuitively, these new social streams can be used to drive automatic event detection systems, potentially providing real-time notifications of emerging events of interest. For example, mobile users might register to be pushed notifications of interesting events happening nearby; similarly emergency services might wish to be notified of fires or crimes as they happen.

Automatic online event detection belongs to a new class of Big Data tasks that have emerged, requiring large scale and intensive real-time stream processing. In particular, these tasks require very high levels of data throughput, while maintaining a low degree of response latency. Indeed, the social Twitter stream generates more than 400 million tweets each day. Other examples of real-time stream processing tasks are stock market trading (approximately 10 billion messages per day in trades) and fraud detection in mobile telephony [6].

However, automatic online event detection on high volume streams is challenging, due to the high computational costs of event detection and the need for very low response latencies. Classical approaches to event detection involve for each incoming document the equivalent of a linear search over all of the documents previously seen, which quickly becomes infeasible as the document stream grows over time. Recent approaches to real-time event detection have improved the per-post processing efficiency, leading to constant time document processing [12]. However, such approaches have so far been limited to single machine processing, resulting in insufficient throughput to process high volume social streams. Scaling such approaches without reducing effectiveness is difficult, since obvious divide-and-conquer approaches, such as partitioning the document stream can result in events being spread across multiple partitions. This reduces the local evidence available when making decisions about whether an incoming post represents a new event or otherwise, resulting in events being missed.

In this paper, we propose a new approach for automatic distributed real-time event detection from high volume streams that can scale efficiently to any volume of input stream, while maintaining event detection performance. This approach uses a novel lexical key partitioning strategy to distribute the computational costs of processing a single document across multiple machines without partitioning the document stream itself. We implement both the proposed approach and a document stream partitioning strategy using the Storm distributed stream processing platform, parallelising a state-of-the-art event detection algorithm [12] to create two distributed event detection topologies. Through experimentation on a large Twitter dataset, we evaluate the effectiveness, efficiency, latency and scalability of the two topologies. Our results show that unlike the document stream partitioning strategy, our proposed approach is able to scale up event detection to process the equivalent of the full Twitter Firehose stream in real-time (4500 tweets each second) without missing additional events (i.e. without degrading effectiveness). Indeed, to the best of our knowledge, this is the first event detection approach that has been shown to be scalable to the full Twitter stream. Moreover, we show that our approach scales close to linearly with processing capacity allocated and maintains low processing latencies.

The contributions of this paper are three-fold. First, we propose a new approach for distributing event detection across multiple machines that is both scalable and tackles effectiveness degradation inherent to stream partitioning strategies. Second, we describe a practical implementation of this approach using Storm for the purposes of real-time event detection on Twitter. Finally, we evaluate the effectiveness, efficiency, latency and scalability of the proposed approach and implementation on the high volume Twitter stream.

II. Related Work

A. Event Detection

The task that we examine in this paper is *event detection*. Event Detection is an application of online clustering, where the objects to be clustered are text documents such as news articles [2] or tweets [12]. Formally, event detection takes as input an unbounded stream of documents, where each document d has a unique id, arrival time and content. It outputs clusters of documents *[d_i...d_j]* representing events. In practice, event detection can be seen as an incremental decision function where a document d is considered to be about a new event if the most similar document previously seen is below a similarity threshold. Documents representing new events can then form the basis of clusters about those events. However, finding the closest document to d is difficult to calculate quickly. Traditional approaches to event detection represent each document d as terms in vector space [1]. The closest document is computed by iteratively comparing d to each other document in the background D. This type of approach has been shown to be effective on low volume newswire streams during the Topic Detection and Tracking workshop [1]. However, comparing against the entire document space becomes infeasible for high volume streams, where the size of D can be in the millions and closest document needs to be found within milliseconds.

Later work by Petrovic et al [12] improved the efficiency of real-time event detection through the use of a locality sensitive hashing (LSH) algorithm [10]. The idea is to approximate the distance to the closest document quickly and in constant time. In particular, they use LSH to group textually similar documents together into clusters, and then compare d to only those other documents in D that were assigned with the same hash key (orders of magnitude fewer documents). The advantage of this approach is that instead of performing $|D|$ document comparisons, each document just needs to be processed by a constant-time hashing algorithm and then compared to each other document with the same hash. On the other hand, while this approach is efficient in comparison to traditional approaches, its throughput on a single machine is far short of that is needed to process high-volume streams such as Twitter. Moreover, as we will show in our later experiments, the natural strategy of partitioning the document stream and deploying this approach on each partition results in reduced event detection effectiveness. It is for this reason that event detection is not embarrassingly parallelisable. Instead, inspired by Petrovic et al's approach, we makes use of hash key grouping to scale up event detection to high volume streams without harming effectiveness.

B. Big Data and Distributed Stream Processing

Traditional data intensive tasks involve the batch processing of large static datasets using networks of multiple machines. To tackle these types of tasks, database management systems (DMBS) [5] and distributed processing frameworks, e.g. MapReduce [7] have proved to be popular. However, it has been shown that traditional DBMSs that use a 'store-then-process' model of computation cannot provide the low latency responses needed for real-time stream processing [3]. Moreover, distributed processing frameworks like MapReduce are not well suited to working with this form of underlying data, due to their batch-orientated nature [7] – leading to a lack of responsiveness [6]. Instead, new distributed stream processing platforms have been proposed, e.g. Storm and S4 [11].

In this paper, we build upon one of these platforms – namely Storm – for real-time automatic event detection. Storm defines computation in terms of data streams flowing through a graph of connected processing instances. These instances are held in-memory, may be replicated to achieve scale and can be run dynamically on multiple machines. The graph of inter-connected processes is referred to as a *topology*. A single Storm topology consists of *spouts* that inject streams of data into the topology and *bolts* that process and modify the data. Topologies facilitate the modularisation of complex processes into multiple spouts and bolts. By connecting multiple spouts and bolts together, tasks can be distributed and scaled.

III. Distributed Event Detection

Scaling event detection systems for high-volume streams is a challenging problem. A common method for distributing stream processing tasks is to separate the high-volume input stream into multiple smaller sub-streams, processing each using different machines. However, in an event detection context, such an approach can degrade event detection effectiveness, i.e. cause additional events to be missed. This is because these strategies spread the documents about an event among multiple event detection instances, meaning that no single instance has sufficient evidence to detect that event. In particular, assume that we have a single stream comprised of 5 twitter posts, three posts about one event and two other unrelated posts. The event detection system will emit a cluster of tweets as an event when it reaches size three. If we process the tweets as a single stream, then the three related tweets will be incrementally clustered together

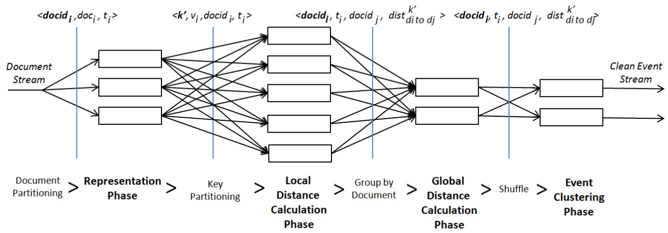


Figure 1. Overview of the four phases of our event detection approach.

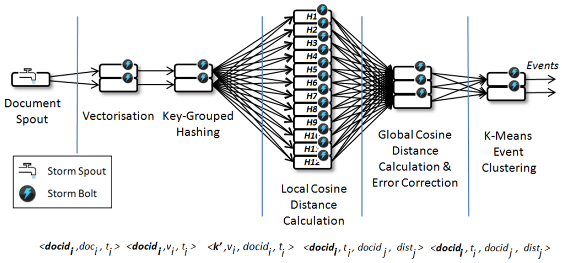

Document Spout
Vectorisation
Key-Grouped Hashing
H1
H2
H3
H4
H5
H6
H7
H8
H9
H10
H11
H12
Local Cosine Distance Calculation
Global Cosine Distance Calculation & Error Correction
K-Means Event Clustering
Events
Storm Spout
Storm Bolt
<docid_i, doc_i, t_i> <docid_i, v_i, t_i> <k', v_i, docid_i, t_i> <docid_i, t_i, docid_j, dist_j> <docid_i, t_i, docid_j, dist_j>

Figure 2. Our approach implemented as a Storm topology.

and then emitted. However, if we partition the stream into multiple sub-streams, then we risk the three tweets being partitioned between the two sub-streams, resulting in fragmentation and reduced evidence for decision making. Hence, we need an more advanced strategy for scaling event detection, which does not compromise effectiveness as parallelism is increased.

We propose a new event detection approach that aims to provide effective scalable low-latency event detection over Big Data streams, while avoiding effectiveness degradation like that described above. Our approach is based on two main concepts. First, inspired by state-of-the-art event detection approaches (see Section II-A), approximate strategies to estimate the distance from the closest document to d should be used to maintain constant time per-document processing. In particular, each document should be represented using one or more lexical keys k and a single spacial representation v. The closest document is then calculated first within local key-partitioned document clusters, reducing the comparison space and avoiding event detection latency increasing over time. Second, the underlying distance estimation for a single document should be partitioned and parallelised across multiple machines, rather than partitioning the stream and parallelising the processing of each subset. This facilitates the scaling of event detection to big data streams without degrading event detection effectiveness. Based upon these two concepts, we propose four distinct processing phases that describe a general event detection approach where each phase can be independently parallelised, namely: *Representation; Local Clustering; Global Clustering*; and *Event Detection*. Figure 1 illustrates each of the four phases of our approach.

A. Event Detection as a Storm Topology

The above approach is generic and can be implemented using a variety of platforms and/or programming languages. For our subsequent experiments we implemented it as a Storm topology as illustrated in Figure 2. Observe that each phase of our approach is implemented as one or more Storm bolts, which can be individually replicated. Each copy of a bolt is referred to as a bolt instance. In particular, the representation phase is spread over two bolts, namely: *Vectorisation* that converts each document into its spacial representation v; and the *Key-Grouped Hashing* bolt that produces (70) hash keys for each document (multiple keys are produced to improve the accuracy of the distance estimation), emitting them as $< k', docid_i, time_i, v >$ tuples. We use locality sensitive hashing for key generation and represent documents in vector-space, as these have been previously shown to be effective for non-distributed event detection [12].

The local distance comparison phase of our approach is represented by the *Local Cosine Distance Calculation* (LCDC) bolt in Storm. The LCDC bolt receives all documents hashed to a subset of the possible hash keys κ. For each hash key $k' \in \kappa$, the bolt maintains a fixed-size first-in-first-out (FIFO) bin of the most recent n (30) documents. When a new document d_i arrives with key k', it is textually compared using cosine comparison to the other documents in the bin for k', returning the identifier of the closest document $docid_j$ and the distance to that document $dist^{k'}_{d_i \rightarrow d_j}$.

Each of the potential closest documents identified are emitted, grouped by their id $docid_i$ and then sent to the global clustering phase of our approach. This phase is represented by the Global Cosine Distance Calculation & Error Correction bolt, which finds the closest of all of the documents (based on the emitted distances). Following [12], this bolt also performs a further error correction function by checking whether any of the most recent m documents are closer that those previously found.

The final phase of our proposed approach; Event Clustering, is represented by the *K-Means Clustering bolt*. This bolt maintains clusters of documents based upon the closest document found. A threshold θ is used to determine whether each incoming document should be added to an existing cluster or should form a new cluster. If a document has been added to an existing cluster, that cluster is emitted as a new event if its size is greater than τ (5). Old document clusters are deleted if they have not been emitted after an hour since their creation. We refer to this topology in our later experiments as *Distributed Lexical Key Partitioning*.

B. Optimisations for Storm

Notably, using a distributed stream processing platform like Storm to process high-volume data streams can incur additional overheads. In particular, each bolt maintains input and output message queues and messages from different bolts on a variety of machines that need to be routed across

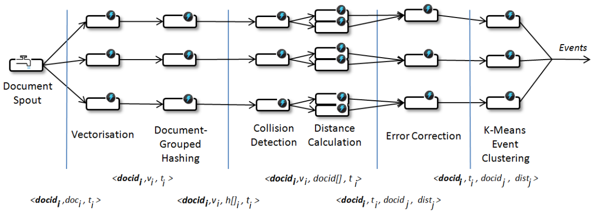


Figure 3. Event Detection using sub-stream partitioning in Storm.

the network and grouped at their destinations. Moreover, since multiple bolts are chained together, a single document will be processed sequentially by each bolt (and possibly by multiple instances of a single bolt as well). As a result, the number of messages traversing a topology processing thousands of documents each second can be in the tens or hundreds of thousands. Hence, to avoid excessive overheads from message passing, we introduce two optimisations to the topology.

First, we introduce buffering and aggregation of messages at the representation phase. In particular, rather than emitting once per key k' produced for a document, we instead buffer documents for each key, emitting them as compressed messages multiple times each second. By producing fewer messages, we can reduce the size of the message queues and enable each message to be better compressed. Second, at the local distance calculation phase, we pre-calculate the global distance calculation instance that each document will be grouped at. We use this to group multiple documents and the calculated distances for those documents together. This optimisation is similar to the role that a combiner plays in MapReduce; reducing the number of emits and hence the time spent sorting/shuffling messages between bolts.

IV. An Alternative Event Detection Topology

Recall that one of the main motivations for our proposed approach was to avoid effectiveness degradation while scaling, which can occur when approaches that partition the document stream are used. In this section, we describe an alternative event detection approach and associated Storm topology that implements such a stream partitioning strategy, which we use as a baseline in our later experiments. In particular, given a high-volume input stream, a natural approach would be to arbitrarily partition that stream into smaller sub-streams. Given an event detection approach with a maximum throughput x, that approach can be replicated and deployed on each sub-stream so long as the rate at which documents arrive on a single sub-stream does not exceed x. The events detected from each effective sub-stream can then be merged into a global event stream as a final step.

Figure 3 illustrates how this approach can be implemented as a Storm topology (assuming the same event detection approach by [12] is used, as before). As we can see, the high volume document stream is partitioned (evenly) into multiple sub-streams. Documents assigned to each sub-stream are converted into their vector representation v within the vectorisation bolt, as before. The vectorised documents are then hashed locally (creating bins local to each sub-stream) within the document-grouped hashing bolt. Each document is then emitted along with all of its hash keys (grouping by document rather than grouping by key). The most recent documents that received the same hash keys are identified and emitted by the collision detection bolt and subsequently compared to find the closest within the distance calculation bolt. This is followed by Petrovic et al's error correction step [12] as a separate bolt. The resultant distance is then used to determine whether an existing local event cluster for the current sub-stream would be updated or a new event cluster for that sub-stream created, as per our proposed approach. All local events are then merged into a global event stream. We refer to this approach in our later experiments as *Sub-stream Partitioning*.

Table I
Comparison of the input and output formats of the two event detection approaches.

Bolt	In/Out	Sub-stream Partitioning	Distributed Lexical Key Partitioning
Vectorisation	In Out	$< docid_i, doc_i, t_i >$ $< docid_i, v_i, t_i >$	
Document-Grouped Hashing / Key-Grouped Hashing	In Out	$< docid_i, v_i, t_i >$ $< docid_i, v_i, k'[], t_i >$	$< docid_i, v_i, t_i >$ $< k', [v_i, docid_i, t_i] >$
Collision Detection and Distance Calculation / Local Distance Calculation	In (1) Out (1)	$< docid_i, v_i, k'[], t_i >$ $< docid_i, v_i, t_i, docid[] >$	$< k', [v_i, docid_i, t_i] >$
	In (2) Out (2)	$< docid_i, v_i, t_i, docid[] >$ $< docid_i, t_i, docid_j, dist_j >$	$< docid_i, [t_i, docid_j, dist_j] >$
Error Correction / Global Distance Calculation	In Out	$< docid_i, t_i, docid_j, dist_j >$ $< docid_i, t_i, docid_j, dist_j >$	$< docid_i, [t_i, docid_j, dist_j] >$ $< docid_i, t_i, docid_j, dist_j >$
Event Clustering	In Out	$< docid_i, t_i, docid_j, dist_j >$ $< docid[] >$	

Table I illustrates the differences between the sub-stream partitioning and our distributed lexical key partitioning topologies in terms of the input and output formats for their bolts. From Table I, we see that sub-stream partitioning differs from our approach in two critical ways. First, the stream is partitioned into multiple sub-streams rather than partitioning on the coarse-grained (hash) key, i.e. the output of the hashing bolt changes from one emit per document in the sub-stream partitioning topology containing multiple hash keys for that document, to periodic emits of multiple documents per hash key under our approach. Second, rather than spreading the distance computation of a single document over multiple machines by having each bolt process individual hashes, documents are processed fully using a pair of bolts, namely collision detection and distance calculation.

V. Experimental Setup

Dataset: To evaluate the effectiveness, efficiency, latency and scalability of event detection, we require a large stream of documents covering an extended time-period covering multiple noteworthy events. Twitter is currently one of the largest social media sources – producing over 160 billion posts each year – and has been shown to be a good source of information about events as they happen [4]. Hence, we use a Twitter dataset containing a sample of 51 million tweets from the period of the June 30^{th} 2011 to

Table II

MAXIMUM PER-BOLT THROUGHPUT AND AVERAGE PROCESSING LATENCY OF THE TWO EVENT DETECTION TOPOLOGIES. † INDICATES THAT THE THROUGHPUT IS ESTIMATED BASED UPON THE NUMBER OF HASHES NEEDED TO PROCESS ONE TWEET.

Topology	Measure	Storm Bolts					
Sub-stream Partitioning		Vectorisation	Document-Grouped Hashing	Collision Detection	Distance Calculation	Error Correction	K-Means Clustering
	Throughput (tweets/sec)	4617.05	623.32	340.45	332.71	266.34	3198.34
	Latency	<0.05ms	<2ms	<5ms	<5ms	<5ms	<0.05ms

Distributed Lexical Key Partitioning		Vectorisation	Key-Grouped Hashing	Local Distance Calculation	Global Dist. Calculation + Error Correction	K-Means Clustering
	Throughput (tweets/sec)	4617.05	563.63	117.07†	414.23	3198.50
	Latency	<0.05ms	<10ms	<100ms†	<5ms	<0.05ms

September 15^{th} 2011 (77 days). All tweets were collected using the public streaming API. From an effectiveness evaluation perspective, although this dataset represents only a fraction of the full Twitter stream, it still covers multiple noteworthy events, including the death of Amy Winehouse and the Moscow Airport Bombing among others. From an efficiency/scalability perspective, high volume streams can be simulated by controlling the rate at which tweets are ingested by the topology.

Events: From the above dataset, we manually identified 27 major events, which we use as the ground truth for our system to detect. For each event, we manually searched the dataset starting from the time when each event occurred with the aim of finding tweets about each. From this search, we found a total of 117,983 tweets over the 27 events.

Hardware: For our experiments, we use a cluster of machines, where each machine contains two 64bit quad-core 2.13GHz processors (8-core), 32GB of RAM, and one 1Tb hard disk. All machines are connected together by a gigabit Ethernet switch on a single rack. Another machine outside this cluster is used for Storm topology control.

Measures: We evaluate the efficiency of our topologies and the individual bolts that comprise them by measuring their throughput in terms of the maximum number of tweets that they can process per second, denoted tweets/sec. Due to small variances in the maximum throughout, we report the average throughput over three runs. Topology scalability is measured in comparison to linear scaling, i.e. when doubling the number of processing cores doubles the throughput. The aim is to achieve as close to linear scaling as possible. Since our topologies are complex and bolts have different throughput rates, the replication factor of multiple bolts need to be increased concurrently to avoid bottlenecking on those bolts that were not replicated. Hence, the linear scaling factor (gradient) that we compare against reflects the number of processing cores needed to increase throughput in a stable manner. Finally, we measure the effectiveness of event detection in terms of event detection recall, i.e. how many of the 27 events were identified by our system. In this case, an event is considered identified if the system emits a cluster that contains 3 or more tweets about the event.

Baselines: We compare our distributed lexical key partitioning approach in terms of effectiveness, efficiency, latency and scalability against the sub-stream partitioning approach described in Section IV. All event detection parameters were set as per the single stream approach that both topologies were based upon [12].

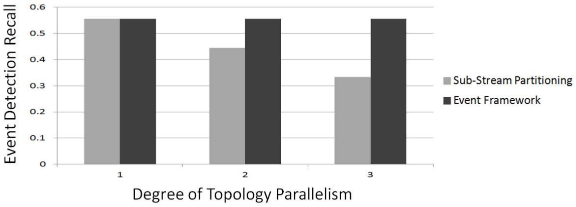


Figure 4. Event detection recall over the 27 identified events for the two topologies at different degrees of parallelism.

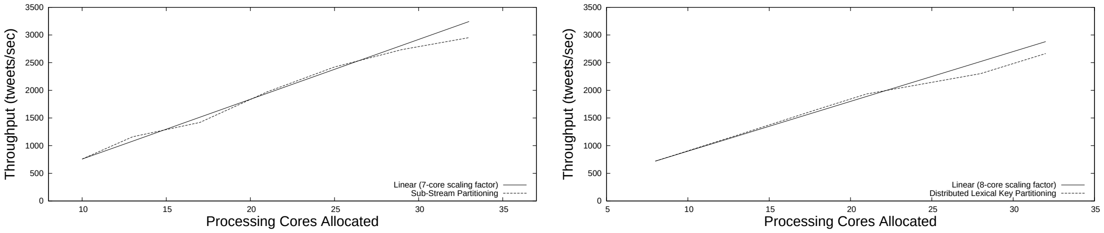
VI. RESULTS

Through experimentation, we investigate three research questions in relation to our approach for real-time event detection, each in a separate section:

- How is effectiveness impacted by parallelisation in comparison to the baseline topology?
- How efficient is our event detection approach in comparison to our baseline topology and are low levels of processing latency maintained?
- Does our approach facilitate event detection that scales close to linearly with processing capacity and can it keep-up with the full Twitter Firehose stream?

A. Effectiveness

We begin by evaluating our first research question, i.e. how is effectiveness impacted by parallelisation under both our approach and the alternative sub-stream partitioning topology. Recall from Section III that we noted that sub-stream partitioning approaches may cause event detection effectiveness to degrade when parallelism is increased, since the aggregate of events detected from parts of the stream are not equivalent to events detected from the whole steam. An effective event detection topology will maintain its performance regardless of the degree of parallelism employed. As such, we evaluate the topology generated using our proposed event detection approach in comparison to the baseline topology described in Section IV.



(a) Sub-stream Partitioning

(b) Distributed Lexical Key Partitioning

Figure 5. Scalability of the two event detection topologies in terms of tweets/sec as we increase the number of processing cores allocated in comparison to linear scaling (using a scaling factor of 7 cores and 8 cores per increment, respectively)

Figure 4 reports the event detection recall of both topologies when as we increase the degree to which the process is parallelised across multiple machines. *N* degrees of parallelism indicates that each bolt within the topology is replicated *N* times. From Figure 4, we see the following. First, that for both topologies, an event recall of approximately 55% is achieved when only one degree of parallelism is used, i.e. 15 of the 27 events in our dataset were identified. Second, we see that as we increase the degree of parallelism, the effectiveness of our proposed approach remains stable, while the sub-stream partitioning topologies effectiveness degrades. Indeed, over 18% absolute recall is lost moving from one to three degree’s of parallelism, which represents only a fraction of the parallelism that we would need to process the entire Twitter Firehose stream (as we show later in Section VI-C). This shows the advantage of using a lexical key-based partitioning scheme for attaining parallelism rather than partitioning over the tweets themselves. Hence, in answer to our first research question, we conclude that our proposed approach is more effective than the baseline approach as it maintains event detection recall while scaling.

B. Efficiency and Latency

Having shown that our approach is more effective than the baseline approach, we next evaluate the efficiency of it in comparison to the baseline topology. Indeed, for a real-time event detection system to be useful, the topology needs to be sufficiently efficient, i.e. such that it can be scaled using relatively few machines. To evaluate how efficient our two topologies are, we simulate high-volume streams by feeding a 1 million tweet sub-sample of our dataset through each at very high input rates (up-to 5000 tweets/sec). We measure the maximum rate at which each individual bolt can process tweets (when no bottlenecks are present), i.e. their maximum throughput over three tests, reporting the average. Table II reports the throughput of each Storm bolt within both topologies. A higher throughput indicates that the bolt is more efficient. Note that for our proposed approach, the data is partitioned by hash rather than by document (tweet), hence we cannot directly measure tweets processed each second for the local distance calculation bolt (where each buffered emit represents one hash and multiple documents). Instead, we report an estimate based upon the number of hashes needed to process a single tweet.

From Table II, we observe the following. First, the throughput of the bolts can vary greatly within a single topology. For instance, under our proposed approach, the fastest bolt was vectorisation; processing over 4000 tweets every second, while the slowest was the local distance calculation bolt, processing just over 100 each second. This is important, since in a deployment scenario we would want to best utilise the available compute resources, i.e. not replicate bolts more than needed to serve the input stream (with some spare capacity to deal with fluctuations in load). Comparing the two topologies, we see that as expected, they differ mainly in terms of the bolts that perform the distance calculations and subsequent error correction (additional distance computations). The sub-stream partitioning topology spreads its computation over three bolts, each able to process between 340 and 266 tweets each second. In contrast, under our proposed event detection approach, the same computation is performed using just two bolts; local distance calculation which is more computationally expensive and global distance calculation, which is less expensive. Overall, the throughput for both topologies are close to equal, although sub-stream partitioning does demonstrate a slight advantage in overall throughput.

Finally, since the aim of event detection is to identify events as quickly as possible, it is important to minimise latency within the topology. Table II also reports the average per-bolt processing latency for a single tweet within each topology. From Table II, we observe that the processing time a tweet spends within each single bolt is between 5ms and 1ms. The exceptions are the key-grouped hashing and local distance calculation bolts from our proposed approach, which have slightly longer latencies (between 10ms and 100ms). This is because these bolts buffer tweets to minimise network traffic, as described in Section III-B. To answer our second research question, both topologies are approximately equivalent in terms of efficiency and maintain very low processing latencies for event detection.

C. Scalability

Having shown that unlike the baseline approach, our proposed approach maintains event detection effectiveness,

Table III
SAMPLE TOPOLOGY CONFIGURATIONS THAT CAN PERFORM EVENT DETECTION OVER THE ENTIRE TWITTER FIREHOSE STREAM.

Topology	Storm Bolts						Total	Throughput
	Vectorisation	Hashing	Collision Detection	Distance Calculation	Error Correction	K-Means Clustering		
Sub-stream Partitioning	2	12	18	18	18	2	**70**	**4864**
Distributed Lexical Key Partitioning	2	14	40		14	2	**72**	**4518**

while using similar levels of processing power. We next investigate our third research question, i.e. is our approach scalable to high volume streams. To this end, we report the maximum overall throughput observed as we increase the number of processing cores allocated to perform event detection. A scalable event detection topology should scale close to linearly with processing power allocated, i.e. doubling the processing cores should also double the topology throughput. The linear scaling factor indicates how many cores are needed to double the throughput from the first data-point. Note that since each bolt in a topology can not achieve the same throughput, it is expected that some bolts may be under-utilised in each configuration. In all cases, the replication factor of each bolt is chosen to minimise under-utilisation (based upon the per-bolt throughputs observed in Section VI-B). Figures 5 a) and b) show the maximum overall throughput of the two event detection topologies in terms of tweets/sec as we increase the number of processing cores allocated in comparison to linear scaling.

From Figure 5, we observe that both topologies scale in a close to linear fashion, i.e. the throughput of both topologies increases in a consistent manner to the number of processing cores allocated. This indicates that both topologies would be suitable for processing of high volume streams (although we showed previously that event detection recall degrades when using the sub-stream partitioning topology). We also observe that the scaling factor of the sub-stream partitioning topology is slightly lower (7) than our proposed approach (8). This indicates that sub-stream partitioning is slightly more efficient, i.e. fewer cores are needed to double the throughput, supporting our observations from Section VI-B.

Finally, to illustrate a practical deployment of these topologies, we perform an additional scaling experiment to determine how many processing cores would be needed to process the entire Twitter Firehose stream (4500 tweets/sec). Table III reports one sample configuration for each of the two topologies and the resultant throughput. From Table III, we observe that approximately 70 processing cores are needed to process the entire Twitter Firehose stream, or approximately 9 8-core machines.

VII. CONCLUSIONS

We introduced a new scalable event detection approach to tackle the difficult task of online event detection using embarrassingly high volume social streams. This approach uses a novel lexical key partitioning strategy to spread the event detection process across multiple machines, while avoiding divide-and-conquer strategies that partition and process the stream as a series of sub-sets. We described an implementation of this approach within Storm, distributing a state-of-the-art event detection system. Through experimentation of a large Twitter dataset, we showed that the proposed approach is able to efficiently scale to big data streams providing thousands of tweets every second, without degrading event detection effectiveness.

ACKNOWLEDGEMENTS

The authors acknowledge financial support for the ReDites Project from EPSRC grant EP/L010690/1.

REFERENCES

[1] J. Allan, V. Lavrenko, and H. Jin. First story detection in TDT is hard. In *Proc. of CIKM*, 2000.

[2] J. Allan. *Topic detection and tracking*. Springer, 2002.

[3] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proc. of SIGMOD/PODS*, 2002.

[4] R. Bandari, S. Asur, and B. Huberman. The pulse of news in social media: Forecasting popularity. In *Proc. of ICWSM*, 2012.

[5] E. Bertino, K.-L. Tan, B. C. Ooi, R. Sacks-Davis, J. Zobel, B. Shidlovsky, et al. *Indexing techniques for advanced database systems*. Kluwer Academic Publishers, 1997.

[6] A. Brito, A. Martin, T. Knauth, S. Creutz, D. Becker, S. Weigert, and C. Fetzer. Scalable and low-latency data processing with Stream MapReduce. In *Proc. of CloudCom*, 2011.

[7] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc. of OSDI*, 2004.

[8] P. Hart. Nearest neighbor pattern classification. *Transactions on Information Theory*, 13(1):21 – 27, 1967.

[9] J. A Hartigan. *Clustering algorithms*. John Wiley & Sons, Inc., 1975.

[10] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proc. of STOC*, 1998.

[11] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *Proc. of ICDMW*, 2010.

[12] S. Petrovic, M. Osborne, and V. Lavrenko. Streaming first story detection with application to Twitter. In *Proc. of NAACL*, 2010.