Efficient Event Detection for the Blogosphere

Patrick Hennig
Hasso-Plattner-Institut
University of Potsdam, Germany
patrick.hennig@hpi.uni-potsdam.de

Philipp Berger
Hasso-Plattner-Institut
University of Potsdam, Germany
philipp.berger@hpi.uni-potsdam.de

Christoph Meinel
Hasso-Plattner-Institut
University of Potsdam, Germany
christoph.meinel@hpi.uni-potsdam.de

Daniel Kurzynski
Hasso-Plattner-Institut
University of Potsdam, Germany
daniel.kurzynski@hpi.uni-potsdam.de

Hannes Rantzsch
Hasso-Plattner-Institut
University of Potsdam, Germany
hannes.rantzsch@hpi.uni-potsdam.de

***Abstract*—In this paper we come up with a novel approach for the early detection of events in blog entries. The detection of trend is already discussed pretty often. Nevertheless, in our understanding the detection of events goes one step further. The presented algorithms detects unique happenings at a given point in time by perceiving unusual frequent occurrences of words or word groups. We introduce an implementation of our algorithm, making use of the SAP HANA database in order to achieve high performance and the ability to answer live queries for events.**

I. Introduction

With the recent increase of social media in the past years, a big amount of user generated content has been created. This data provides many interesting research opportunities. In this paper we are focusing on the detection of significant changes in this corpus and come up with an approach to summarize changes to events.

Event detection algorithms could be used for disaster recognition. Sakaki et al. used Twitter in their work [1] to detect earthquakes before the disaster prevention could recognize them. Further usages of event detection are automatic news creation or support for news reporters and personalized advertisement.

In this paper we introduce a new approach to event detection for blogs, which is one important category of social media, in which continuously large amounts of data is being generated. The proposed algorithm was implemented in the context of the *BlogIntelligence*[1] project at the Hasso Plattner Institute, Potsdam and tested with real world data collected cumulatively during the time of the project's duration.

As there is no consistent and formal definition of the term *event*, we propose the following definition: An Event is a unique happening at a given point in time or a time interval. Each event is identified by a small set of *keywords*. A keyword in turn is defined as a word or word group which is representative for a document, in our case a blog post.

Consider a campaign speech of German chancellor Merkel as an example for an event. A possible subset of the keywords describing this event might be "Merkel", "Berlin", and "campaign speech".

In contrast to other implementations we laid the focus of our work to the performance of the algorithm's implementation. One of our goals was the ability to answer live queries. One challenge we faced concerning this requirement was the vast amount of data provided by the BlogIntelligence project. About 80 millions of blog entries containing 3 million unique keywords had to be considered for the detection of events. In order to make use of this huge amount of data, the project is using an in-memory database provided by SAP, called *SAP HANA*[2] as discussed by Hennig et. al [2] . Our implementation enables the execution of the computation in a highly parallelized environment. The computation of the events is integrated into one of the deepest levels of the database. Thus, keeping the amount of data transferred between the different levels of the database is very minimal.

A. Project Scope: BlogIntelligence

With a wide circulation of more than 200 million *weblogs* worldwide, *weblogs* with good reason are one of the most important data streams in the World Wide Web. Therefore, weblogs offer access to latest information discussed in the real world. Since writing posts in weblogs goes along with a high editorial effort, the available information is of major interest. However, for a user it is becoming harder and harder to gain an overview of all discussions in the blogosphere. Hence, a system that collects information from the blogosphere and presents it to the user in a very meaningful way would be of great use.

Therefore, mining, analyzing, modeling and presenting this enormous amount of data is the overall aim of the project the presented work is integrated in. This enables the user to detect technical trends, political climates or news articles about a specific topic. Most approaches to mining and analyzing such a huge amount of data focus on offline algorithms which use pre-aggregated results. This is in contrast to the continuously growing nature of the World Wide Web. As a result, including the latest data is one of the key aspects of data mining on the web. This is exactly the topic covered by the *BlogIntelligence* project.

The presented work in this paper is integrated into the *BlogIntelligence* project. There are three main steps involved to visualize blogs in the BlogIntelligence project:

[1] http://www.blog-intelligence.com

[2] http://www.saphana.com

1) Extraction: In the extraction step the blogs are basically crawled. In order to achieve this a, purpose-built crawler needs to be used as traditional crawlers do not fully meet the particularities of blogs as opposed to conventional websites.

2) Analysis: The analysis step prepares the crawled data for visualization. Each blog is analyzed by multiple *Analyzers*, that process its details in certain ways. Among potentially others, there are *data analyzers* that store the meta information about the blogs into the database, *content analyzers* that store information about the content which allow content-related analyses and there are *network analyzers* that store information on the relationships and links between blogs or other communities.

3) Visualization: The last and very important step within the BlogIntelligence framework is the visualization of the analyzed information. Hereby, new ways of visualization of this big amount of data from the social web is being tested.

II. Related Work

The results of the research of Sayyadi et al.[3] and Fung et al.[4] form the foundation of our work. Both groups worked under the assumption that documents which describe the same event contain similar sets of keywords. Therefore, the automatic detection of events tries to identify such sets. The researchers proposed different approaches to discover these sets.

Sayyadi suggests extracting keywords from documents and clustering them based on their co-occurrence in the documents. They consider keywords, that are clustered together, to belong to the same event. Unfortunately, the proposed algorithm requires a very accurate selection of many parameters and thresholds. These values have a large impact on the quality of the generated results and are very hard to determine due to their inter-dependencies.

Tackling this problem, Fung proposed a time based clustering approach. Thus, fewer parameters have to be set manually. This technique provides the basis for our algorithm. In contrast to Sayydi's approach and building on the work of Fung, our algorithm takes the time interval, in which a keyword occurs, into account. This avoids the effect of merging events with similar keyword sets but different occurrence times to one larger event.

Both of the research teams discussed above made use of data sets containing only news articles or blog posts referring to news. Hence, they work on preselected data that is known to deal with events. Our implementation however works on a data set where news articles are only a minority of the documents, while the majority of the documents does not describe events on their own. Instead, we aggregate information from many documents to detect events even from these documents.

A different event detection mechanism was shown by Jurgens[5]. The group used changes of keyword semantics over time periods to detect events.

III. Concept

In this section we give an overview of our algorithm. We explain the derivation of each algorithm step and present the resulting algorithm. All steps are illustrated in Figure 1.

Given a set of documents and a time interval we want to determine a set of events, that is described by these documents and took place during this time interval. Each document represents a blog post and provides an ID, the blog text and its publication date. The underlying assumption is that an occurring event generates a large number of documents reporting on this event. For example, after the election of Angela Merkel there were many blog posts reporting on the election and its result. Therefore, we analyze the content of blog posts.

We use keywords as content representation of documents. Keywords are defined as words or word groups which are representative for documents. Therefore, in the first step we extract one list with all keywords from all documents. Since events are also represented by keywords they are the connection between events and documents.

We consider unusual high occurrence frequencies of a keyword as an indicator for an event. As mentioned previously an event results in a high count of documents reporting on this event. As a result, the frequency with which keywords occur in these documents is higher than usual. Thus, we analyze the frequency distribution over time of each keyword in the second step. In the third step we detect the time intervals in which these frequencies are unusually high. We call these intervals *bursty* in accordance with established literature [4].

We detect Keywords that are bursty at the same time and occur in the same documents as events. The keywords of an event are bursty at least in the time interval of the event. Therefore, finding all bursty keywords in a time interval we find all events happening in this time interval. Dividing these keywords into sets of keywords in a way that all keywords of a set belong to the same documents, these sets resemble the events. Clustering keywords to events is our fourth step.

To provide more information about events than only keywords, we find related documents for each event. By analyzing the documents of an event we could extract more information such as a location, the participants and the importance in future. But for now, we only use this information to sort the events by the number of documents the event is related to.

IV. Implementation

This section focuses on the most important implementation details of the process described in Figure 1.

A. Extraction of keywords

The extraction of keywords aims to find a small set of descriptive words for the document. Therefore, we need a means to evaluate the importance of a word for the respective document. Various approaches exist in order to achieve this goal. The two techniques our algorithm makes use of are the TF-IDF metric[3] and the Named-entity Recognition[4] (NER).

The first method provides an easy and fast way to identify words that are often used in a document and rarely used in other documents. The weakness of this approach is that each word will be treated separately from others if the text was only

[3] see [6] for an overview to the usage of TF-IDF

[4] see [7] for an overview to NER

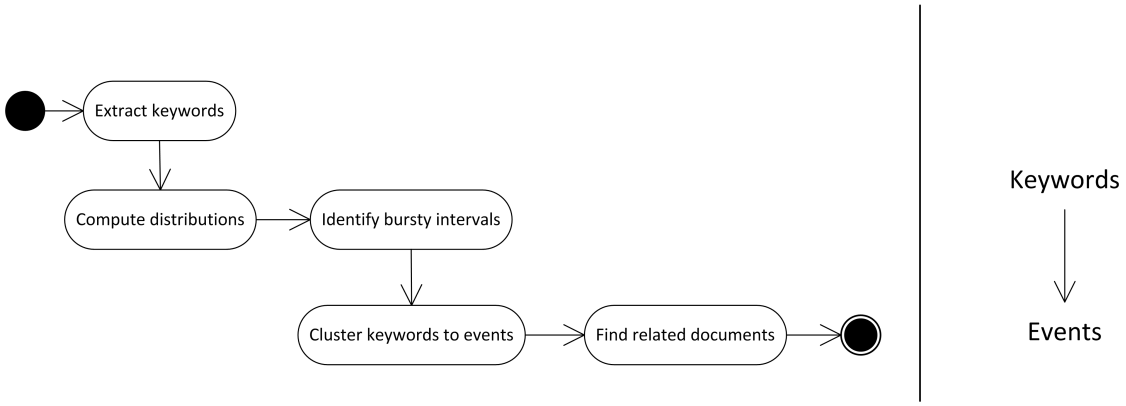

Fig. 1: Abstract algorithm steps

tokenized before by white spaces. While TF-IDF provides a simple and fast way to identify words that occur frequently in a document while occurring rarely in other documents, its weakness is that every word is treated separately. This can be problematic, for example when evaluating people's first and last names, which are identified as separate keywords. This leads to corrupted results in the following algorithm steps. The use of NER—treating first and last name as one term—solves this problem. After the NER we still need to calculate the TF-IDF values for the resulting terms. This way, we get a global metric for a term's relevance in all documents. Note that instead of NER also n-gram[5] could have been used here.

In our system, the in-memory database performs the NER and provides all terms for each document. As described above, these terms can consist of multiple words, if the words form one entity for the NER. Therefore, we are able to apply the TF-IDF metric on these terms. If the TF-IDF result exceeds a predefined threshold the term will be classified as a keyword for the document. The computation of the TF-IDF values is also done by the database.

The TF-IDF computation and therefore the extraction of keywords depends on the given time interval. Therefore, only documents published in this time interval are considered for the TF-IDF computation. This limitation is important for the algorithm's efficiency.

B. Computation of keywords' distributions

The second step is the computation of the occurrence frequency over time for each keyword. As in the previous step, we only compute the distribution for the given time interval and apply the same limitation to the documents.

In order to count the occurrences of keywords, we need to build buckets depending on the time parameter. Otherwise, a keyword will rarely occur more than once at the same time, because the publishing dates of documents are given exactly to the second. Therefore, we sample the time interval in time frames. The length of each frame is constant and called *time unit*. For example, a time unit can be an hour or a day. Resizing the time unit allows for control over the tradeoff between accuracy depending and computation costs.

The input for the distributions' calculation comprises the keyword, the documents, and the time interval. The idea behind this is to compute the occurrence value of the keyword for each time unit in the given interval and to normalize this value, so it can be compared to other values later. The normalization is needed because the number of documents published differs between different time units. Therefore, the absolute number of occurrences does not provide a reliable basis of comparison. To improve the performance of the next algorithm step it is important to provide the occurrence history sorted by time. The pseudocode for the keyword frequency distribution calculation is illustrated in Algorithm 1 on page 4.

First the start and end time values of the given time interval are adjusted depending on the length of a time unit. The new start value is set to the begin of the time unit the old value belongs to, the new end value to its end. This time interval adjustment allows us to divide the given interval into whole time units, which is important for the normalization step.

After the time interval adjustment, the algorithm iterates over all time units in the redefined interval starting with the earliest one. In each iteration, the set of documents published in the currently viewed time unit will be determined. For sorted or indexed data, this step performs very efficiently.

The normalized occurrence value for a time unit can be calculated by dividing the number of documents encountered in the last step and containing the given keyword by the number of all documents relating to the time unit. If the resulting value is non-zero we store it in the result list together with the respective time unit. The resulting values are also sorted by time units due to our choice of the iteration order. This also saves us the computational expense for sorting them.

The provided algorithm needs only read access to the document corpus. The distribution computation of one keyword is independent of other keywords. Therefore, the distributions can be computed easily in a parallel way and without additional need for synchronization.

[5]consult [8] for an introduction to natural language classification using n-gram models

Algorithm 1 Calculation of occurrence histories

```
function KeywordHistory(keyword k, document corpus D, time interval t)
    result ← empty list
    ▷ Redefine time interval begin
    start ← t.start − t.start%TIME_UNIT
    ▷ Redefine time interval end if needed
    if t.end%TIME_UNIT ¿ 0 then
        end ← t.end − t.end%TIME_UNIT + TIME_UNIT
    end if
    ▷ Iterate over all time units in given interval
    for time tc ← start; tc < end; tc ← tc + TIME_UNIT do
        relevantDocs ← {d ∈ D : tc ≤ d.publicationDate ≤ tc + TIME_UNIT}
        frequency ← 0
        ▷ count the number of documents containing k
        for document d ∈ relevantDocs do
            if count(k, d) > 0 then
                frequency ← frequency + 1
            end if
        end for
        ▷ Normalize the frequency
        frequency ← frequency/|relevantDocs|
        result.add((tc,frequency))
    end for
    return result
end function
```

C. Identification of busty intervals

During this step of our algorithm we determine bursty intervals in the distribution of each keyword. Bursty intervals describe time intervals in which the keyword occurs unusually often. Statistical metrics like the expected value μ of occurrences and the related sample standard deviation s are used to define which occurrence frequencies are unusually high and which are not. Assuming a normal distribution of keywords, we define all occurrence frequencies larger than a threshold of $\mu + s$ as unusually high. This approach leads to good results that are comprehensible for humans.

It is important to note that these statistical metrics are not computed for the whole distribution period but for smaller parts of this distribution, termed *subintervals*. Otherwise, long periods of very low occurrence frequencies that usually occur would falsify the detection of burstiness. This solution to the problem was proposed by Fung [4].

Unfortunately, another problem results from this solution. Subintervals with constant or nearly constant occurrence frequencies are always classified as bursty intervals because the standard deviation is close to zero and the expected value is almost equal to each occurrence frequency in the subinterval. To tackle this problem we introduce an additional threshold, constant for each subinterval. For this additional threshold we use the expected value of the whole occurrence history. Thereby we are able to find time units where the keyword is bursty. We identify bursty intervals by merging adjoining time units. Figure 2 on page 5 illustrates this procedure.

In order to implement this procedure, a way to compute subintervals needs to be determined. In our system, we use constant subintervals and quantify the length of a subinterval in time units. For example, if we define a day as a time unit, an appropriate subinterval length is a week. For an hour as a time unit, we propose a subinterval length of a day.

After choosing the subintervals' lengths, we now have to choose when each subinterval should start. Fung proposed to split the whole time interval into subinterval by having the first subinterval start at time 0, the second at the end of the first one and so on. As the results of this approach are often not intuitive for humans, we pursue a different approach. In contrast to the work of Fung, we decided to test each possible subinterval which could be appropriate for a time unit. Because each subinterval consist of whole time units, we can easily check each possibility for the start time. This method is an adaption of the moving windows approach and provides more smoothed and human intuitive solutions than the approach proposed by Fung.

The computation of bursty intervals depends only on the distribution of the respective keyword. Therefore, the bursty intervals for different keywords can be computed simultaneously and without additional synchronization.

D. Creation of events

For the final creation of events we use the extracted keywords and the related bursty intervals to identify events for the given time interval. We defined an event as a set of keywords that are bursty in the same time interval. To build this set we cluster keywords with overlapping bursty intervals. As mentioned in Chapter III the time interval cannot be the only feature for our clustering algorithm, because different events could occur at the same time. Instead, we use not only the time of keywords' occurrences but also co-occurrences and conditional co-occurrences of keywords as it has been proposed by Sayyadi[3].

Two keywords k_1 and k_2 co-occur if they are used in the same document d. We define the co-occurrence value of two keywords in a time interval as the number of documents, published in the given interval and containing both keywords divided by the total number of documents $|D|$ published in the interval. The co-occurrence value provides a possibility to decide how often both keywords are used together. The higher this value is, the higher is the relevance of the relationship between these two keywords. But this value is not enough to decide on the quality of such relationships, only about the quantity.

We use the conditional co-occurrences to decide on the quality. The conditional co-occurrence values describe how often two keywords co-occur if one of them occurs. As a result of this definition, there are always two conditional co-occurrence values for each pair of keywords With these two values we can easily decide on the quality of the relationship. If one of these values is low then the keywords not belong to each other. The formulas in Figure 3 show the mathematical definition of both metrics.

Based on the two metrics, co-occurrences and conditional co-occurrences, Sayyadi proposes an approach to cluster keywords. The keywords and their relationships are represented

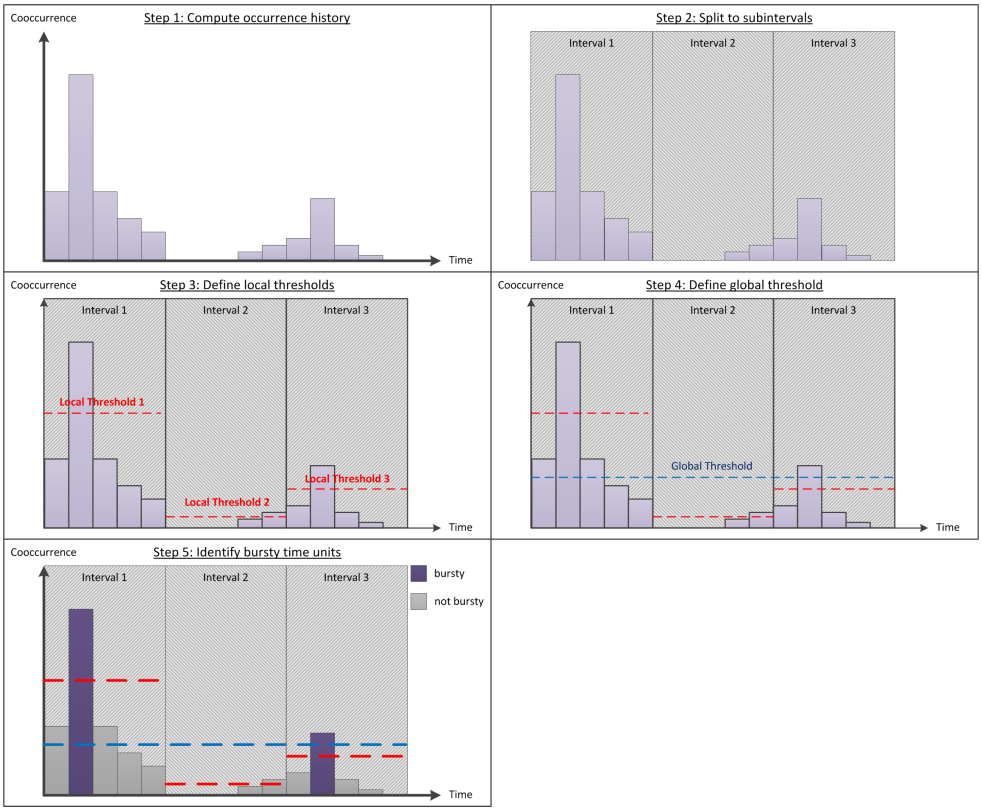

Fig. 2: Procedure for the identification of bursty time units

$$\text{co-occurrence}(k_1, k_2) = \frac{|\{d \in D \mid k_1 \in \text{d.text} \ \wedge \ k_2 \in \text{d.text}\}|}{|D|}$$

$$\text{conditional co-occurrence}(k_1|k_2) = \frac{|\{d \in D \mid k_1 \in \text{d.text} \ \wedge \ k_2 \in \text{d.text}\}|}{|\{d \in D \mid k_2 \in \text{d.text}\}|}$$

Fig. 3: Definition of co-occurrence and conditional co-occurrence

as a graph. In the so called *KeyGraph* each node represents a keyword and each edge represents the relationship between the keywords corresponding to the nodes. The idea behind Sayyadi's method is to identify highly connected sets of nodes and to define each such node set as an event.

To achieve this, Sayyadi proposes to identify edges which are important for the connectivity of the KeyGraph and to remove them. The resulting graph consists of independent subgraphs. Figure 4 on page 6 demonstrates a simplified example. Sayyadi uses the betweenness centrality measure[9] to identify edges which have a high connectivity. From our point of view this approach fails to take the time parameter into account. Furthermore, such KeyGraphs tend to be large. In our experiment with a small dataset of 1000 blog posts, we extracted 5400 keywords and each keyword must be represented as a node in the KeyGraph. The main problem with the analysis of such large graphs is the computation of the betweenness centrality measure. The known algorithms for computation of the betweenness centrality on directed and weighted graphs are not very efficient. For example, the Floyd-Warshall algorithm has a complexity of $O(n^3)$, where n is the number of nodes.

For these two reasons, we adjusted Sayyadi's approach to solve both problems. First we introduced the time parameter dependency into the clustering method. We split the time interval that should be searched for events into time units and generate KeyGraphs for each time unit. The KeyGraph for a time unit then contains only keywords that are bursty in this time unit. Now we can generate a set of events for each KeyGraph and unify the resulting sets to one result. This approach allows to take the time parameter into consideration and to shrink the size of the individual KeyGraphs drastically. The performance gain from this step depends on the keyword distribution over time and the length of the search interval. For a common data set where each time unit has the same amount of bursty keywords, the KeyGraphs shrink by the factor of time units in given the search interval.

But this solution causes a new problem: Events that occur during more than one time unit are split to different events. Therefore, we must check all resulting events during the union step and merge split events together. Two events are similar enough to be merged if they are described by similar keyword sets and occur closely one after another.

We solved the problem of introducing the dependency from the time by splitting the KeyGraph into one KeyGraph per time unit. The second problem, the inefficient computation of the betweenness centrality, remains to be solved. We propose

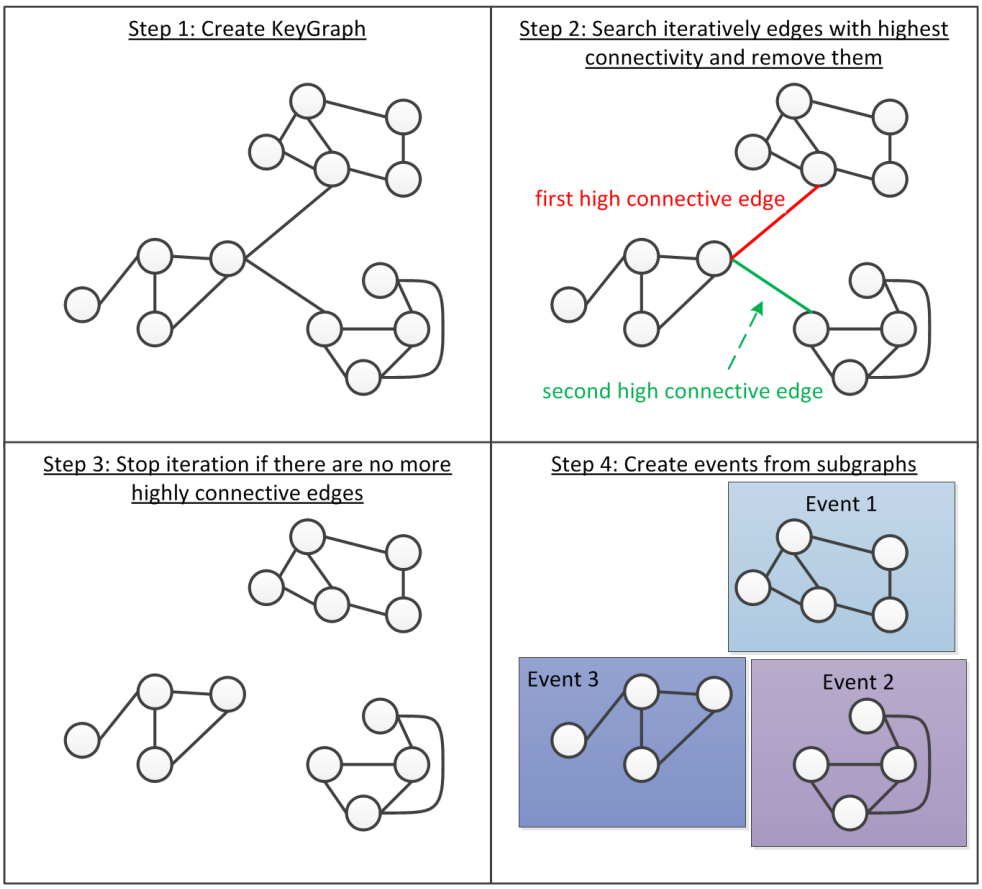

Fig. 4: Clustering of keywords with the KeyGraph

different heuristics which could tackle this problem.

First the average KeyGraph size should be reduced. As proposed by Sayyadi, we remove all edges for which either the co-occurrence or conditional co-occurrence values between the related keywords lie under a threshold. These edges represent relationships with too low relevance for the event detection or with too low quality of the relationship. We observed in tests with real world data that the resulting KeyGraphs often consist of many smaller KeyGraphs, which are not connected to each other. Therefore, we can split such KeyGraphs in linear time into smaller graph structures and analyze them much faster than one large KeyGraph.

A further improvement is the use of the Dijksta algorithm for the computation of the betweenness centrality measure, since it is easy to parallelize and achieves better performance in sparse graphs than the Floyd-Warshall algorithm.

In the work [10] Brandes proposed a probably faster way for computations of the betweenness centrality in undirected graphs. Due to the limitation to undirected graphs we did not implement this approach. For future research the relation between the achieved speed-up and the loss of accuracy could be examined, though.

Algorithm 2 on page 7 shows the cooperation between the approach steps described above. We used a stack structure to avoid recursion and to realize the depth search, to avoid memory problems.

This step is easy to parallelize. The search of events for one time unit is completely independent from other time units. This fact allows us to compute the events for each time unit simultaneously. In contrast to previous steps, we need to merge the results of the parallel computations. To avoid synchronization overhead, we provide another result set to each of the parallel computations and merge them after all computations are finished.

E. Matching of documents to events

After the creation of events, we need to find documents that relate to the events in order to provide the user with a proper description of the created events.

By limiting the set of documents to only those which were published in the time interval of the event, we reduce the number of needed comparisons massively. As mentioned earlier, this reduction step performs very efficiently for sorted or indexed data.

Now we are able to compare the documents with events based on the keyword sets both object types provide. Each document has a set of keywords which are used in the document text. Each event is defined by a set of keywords. By computing the similarity between these two sets, we are able to decide how well a document represents an event.

Algorithm 2 Algorithm for event creation

```
procedure FINDEVENTS(IN TIME_INTERVAL TI, OUT EVENTS E)
    for TIME_UNIT TU ∈ TI do
        EVENTS E_TU ← {}
        FindEvents(TU, E_TU)
        MergeEventsFromTo(E_TU, E)
    end for
end procedure

procedure FINDEVENTS(IN TimeUnit TU, OUT EVENTS E)
    KEYGRAPH k ← createGraphFor(TU)
    STACK S ← {k}
    EVENTS E ← {}
    while NOT empty(S) do
        KEYGRAPH sTop ← pop(S)
        EDGE e ← bestSplitEdge(sTop)
        if valueBC(e) ≥ threshold then
            removeEdge(e, sTop)
            if splitNeeded(sTop) then
                KEYGRAPHS graphs ← split(sTop)
                pushAll(graphs, S)
            else
                push(sTop, S)
            end if
        else
            addAll(eventsFrom(sTop), E)
        end if
    end while
end procedure

procedure MERGEEVENTSFROMTO(IN EVENTS toMerge, INOUT EVENTS toMergeInto)
    for EVENT eventToMerge ∈ toMerge do
        for EVENT eventToMergeInto ∈ toMergeInto do
            if similar(eventToMerge, eventToMergeInto) then
                merge(eventToMerge, eventToMergeInto)
                break
            end if
        end for
        if noSimilarEventsFound(eventToMerge) then
            add(eventToMerge, toMergeInto)
        end if
    end for
end procedure
```

For the comparison of keyword sets we used the well-known Jaccard index, because it can be implemented very efficiently with our data structure. The cosine similarity measure could also be employed for this task and provides similar results as the Jaccard index in our case.

To evaluate the relevance of the created events, we sort the events by the number of documents that belong to them and filter out events with too few documents.

This algorithm step is easy to parallelize. The comparison of an event and a document is independent from other comparisons and needs only read access to keyword sets of both objects. Therefore, we can simultaneously compare events and documents.

V. EVALUATION

Since it was one of our main goals to allow the detection of events to be run in a highly parallized way, we want to provide some evaluation here. In the following we will discuss each step of the algorithm regarding its scalability.

Figure 5 shows the performance of the identification of bursty intervals, the creation of events, and the matching of documents to events respectively[6].

The table does not provide measurement data about the extraction of keywords as the keywords are obtained directly from the database. As the table about the performance of the event creation step reveals, there has been a problem scaling this part of the algorithm. This undesired behavior results from a too small caching capability and thus many context switches on the machine we used for the measurements. It does not occur on the productive system of the BlogIntelligence project.

The overall performance of the algorithm also depends on the number of keywords we extract from each document. The number of keywords directly influences the size of the KeyGraph described in section IV-D. Hence, the computational cost for the creation of events is increased proportionally.

Another factor that impairs the algorithm's performance is unclean data. The algorithm can misinterpret non-content strings—such as html tags—that have not been stripped from the crawled documents as keywords. For the creation of events from bursty keywords, those strings are of no value.

VI. CONCLUSION

In this paper we introduced aa approach for the detection of events in blogs. The algorithm we presented is geared to high performance, sufficient for answering live queries. As it has been pointed out in section V our implementation processes input documents in a parallelized manner with multiple threads. Also, the implementation is embedded deeply into the database source, making optimal use of its hardware capabilities.

Speaking from an algorithmic point of view, our implementation combines the idea of a co-occurrence based keyword clustering algorithm with a time based approach to clustering. Thus, we achieve very good clustering results while avoiding a large set of user defined parameters as well as the effect of undesired merging of temporally separated events.

VII. FUTURE WORK

Since the event detection software is meant to be run as part of the BlogIntelligence project it would be desirable to create a precise mechanism to measure the results' quality. Therefore, a tagged data set would be necessary, which provides ground truth about the events that should be detected by the algorithm. Currently there is no such data set available. Hence, the data set needs to be created.

[6]The event detection was run on a data set of about 10000 documents (about 20 MB) on a Windows 7 machine, Intel Core i7 Q720, 1,6 Ghz, 8 GB Ram

Threads	Runtime
1	29952
2	15538
4	10467
8	7987

(a) Computation of keyword distribution and identification of bursty intervals

Threads	Runtime
1	4649
2	4430
4	5429
8	6427

(b) Creation of events from keyword clustering

Threads	Runtime
1	514
2	281
4	250
8	203

(c) Matching of documents to events

Fig. 5: Performance of each step of the algorithm in dependence of the number of threads employed

There are also yet not implemented means to improve the quality of the generated events. In contrast to plain text streams there is much meta information associated to blog entries. Among other data, this includes the category of the blog and the location where it originates from. How well this information can be employed in order to identify blog entries discussing the same event has not yet been researched.

In addition to using the meta information for improving the quality of the events, the information could also be used to provide more details about the events. Thus augmenting events with information about groups of persons associated to the event or locations which are relevant to it.

Finally, *stories* of events could be created by identifying events that have similar keyword sets but take place at different times. In many cases multiple events belong to one larger context. Thus, they could tell a story if this relationship was unveiled. Consider the release of a new smart phone as an example: Typically, the phone would first appear in the media when some information about it is "leaked". A few weeks later, the phone will be officially announced. After that it will be presented to the public and finally it will be released to the stores. While each of these four points in time marks a single event, they tell one story once we look at them together.

References

[1] T. Sakaki, M. Okazaki, and Y. Matsuo, "Earthquake shakes twitter users: Real-time event detection by social sensors," in *In Proceedings of the Nineteenth International WWW Conference (WWW2010). ACM*, 2010.

[2] C. M. Patrick Hennig, Philipp Berger, "Web mining accelerated with in-memory and column store technology," in *Proceedings of the 9th International Conference on Advanced Data Mining and Applications (ADMA 2013)*. Springer-Verlag Berlin Heidelberg 2013, 12 2013, pp. 205–216.

[3] H. Sayyadi, M. Hurst, and A. Maykov, "Event detection and tracking in social streams," in *In Proceedings of the International Conference on Weblogs and Social Media (ICWSM 2009). AAAI*, 2009.

[4] G. P. C. Fung, J. X. Yu, P. S. Yu, and H. Lu, "Parameter free bursty events detection in text streams." in *VLDB*. ACM, 2005, pp. 181–192. [Online]. Available: http://dblp.uni-trier.de/db/conf/vldb/vldb2005.html#FungYYL05

[5] D. Jurgens and K. Stevens, "Event detection in blogs using temporal random indexing," in *Proceedings of the Workshop on Events in Emerging Text Types*, ser. eETTs '09. Stroudsburg, PA, USA: Association for Computational Linguistics, 2009, pp. 9–16. [Online]. Available: http://dl.acm.org/citation.cfm?id=1859650.1859652

[6] J. Ramos, "Using tf-idf to determine word relevance in document queries," in *Proceedings of the First Instructional Conference on Machine Learning*, 2003.

[7] A. Borthwick, "A maximum entropy approach to named entity recognition," Ph.D. dissertation, New York University, 1999.

[8] P. F. Brown, P. V. deSouza, R. L. Mercer, V. J. D. Pietra, and J. C. Lai, "Class-based n-gram models of natural language," *Comput. Linguist.*, vol. 18, no. 4, pp. 467–479, Dec. 1992. [Online]. Available: http://dl.acm.org/citation.cfm?id=176313.176316

[9] L. C. Freeman, "A set of measures of centrality based on betweenness," *Sociometry*, pp. 35–41, 1977.

[10] U. Brandes, "A faster algorithm for betweenness centrality," *Journal of Mathematical Sociology*, vol. 25, pp. 163–177, 2001.