Mining large distributed log data in near real time

Stefan Weigert
TU Dresden
Dresden, Germany
stefan@se.inf.tu-dresden.de

Matti Hiltunen
AT&T Labs Research
180 Park Ave.
Florham Park, NJ, USA
hiltunen@research.att.com

Christof Fetzer
TU Dresden
Dresden, Germany
christof@se.inf.tu-dresden.de

ABSTRACT

Analyzing huge amounts of log data is often a difficult task, especially if it has to be done in real time (e.g., fraud detection) or when large amounts of stored data are required for the analysis. Graphs are a data structure often used in log analysis. Examples are clique analysis and communities of interest (COI). However, little attention has been paid to large distributed graphs that allow a high throughput of updates with very low latency.

In this paper, we present a distributed graph mining system that is able to process around 39 million log entries per second on a 50 node cluster while providing processing latencies below 10 ms. We validate our approach by presenting two example applications, namely telephony fraud detection and internet attack detection. A thorough evaluation proves the scalability and near real-time properties of our system.

Keywords

Log processing, distributed graphs, COI

1. INTRODUCTION

The volume of log data in complex distributed systems can become very large due to the amount of data generated by individual nodes (e.g., a network router generating netflow data) as well as the number of nodes in a complex system (e.g., 100K+ compute nodes) each generating log data. Processing and analyzing this data to identify events of interest or to store the data for user initiated queries becomes a challenging task. This task is particularly challenging if the data needs to be analyzed in real time, for example, applications such as fraud detection require new entries in the log files to be analyzed in less than a second. Given the volume of data, it is typically not possible to store all the required data in the memory of one processing node.

Graphs are used in various applications to process log-data. Examples include clique analysis [11], query-log analysis for search-engines [9], graph databases [20], pattern matching [21], and fraud detection [7, 19]. Graphs can generally be used to store data representing interactions between entities such as phone calls between calling and called phone numbers, Internet communication between two end points (each defined by IP address and port number), or interactions between software components (e.g., call graph).

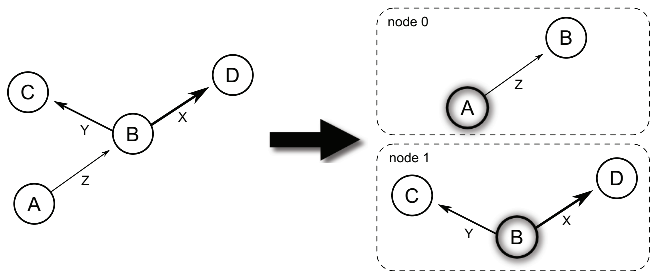

Figure 1: Distributed graph

In this paper we will demonstrate that large graphs can be updated and queried in near real time by distributing the graph onto multiple nodes (i.e., physical machines). We show that the space complexity for the distributed graph is linear.

Our contributions can be summarized as follows:

1. We construct dynamic, distributed graphs with linear space complexity.
2. We make these graphs easily accessible for queries.
3. The system scales linearly with the number of processing nodes.
4. We avoid unnecessary data copies by using the log-generating nodes for processing.

The rest of the paper is structured as follows. We describe the approach and architecture in Section 2, followed by two example applications (telephony fraud detection and internet attack detection) in Section 3. We evaluate the performance of the system with these example applications in Section 4. Section 5 provides a survey of related work, followed by conclusions in Section 6.

2. APPROACH AND ARCHITECTURE

2.1 Distributed graph

We propose a distributed, dynamic graph structure as depicted in Figure 1 as means for log processing. The graphs are directed, potentially cyclic, and do not need to be fully connected (i.e., there may be a vertex v_1, not reachable from a distinct vertex v_2). With this structure, correlations and relationships can be expressed easily. For example, the vertices A, B, C, and D could be physical machines, sending error messages X, Y, and Z to each other. This information could be used to determine how an error propagated through a complex system.

The graphs consist of a set of vertices V and a set of edges E: $G = (V, E)$ with $E \subseteq \{(v_1, v_2) \mid v_1, v_2 \in V\}$. Each edge $e \in E$ has a weight $w(e) \in \mathbb{R}$ and can store any kind of additional information (e.g., the different ports used, for every observed communication between v_1 and v_2).

2.1.1 *Sub-graph construction*

To distribute such a graph onto multiple nodes (i.e., physical machines or separate address spaces), we construct for each vertex $v \in C$, a sub-graph $G_v = (V_v, E_v)$ with:

$$V_v = \{v_a \in V \mid \exists e \in E \wedge e = (v, v_a)\} \cup \{v\}$$

where the set of vertices C contains only those vertices $v \in V$ which have at least one outgoing edge. The new set of vertices V_v now contains all vertices v_a that are connected with an edge from v to v_a, as well as v itself. To construct the set of edges E_v we add all outgoing edges $e \in E$ of v:

$$E_v = \{e \in E \mid \exists v_1 \in V : e = (v, v_1)\}$$

The vertex v is now called the center vertex of the sub-graph G_v. To determine on which node to store sub-graph G_v, we apply a hash-function h to the center vertex v and compute the modulo of $h(v)$ with the number of available nodes. Since each node has a unique id, which starts at 0, the result of the modulo operation equals the unique id of the node on which to store G_v. For example, in Figure 1, $h(A) \bmod 2 = 0$ matches the unique id of *node 0* and $h(B) \bmod 2 = 1$ matches the unique id of *node 1*.

2.1.2 *Space complexity*

Because the sum of all $|E_v|$ is equal to $|E|$, the space needed to store all edge information is not changed by the sub-graph construction. The transformation does, however, increase the space needed to store the vertices, that is, the sum of all $|V_v|$ is greater or equal to $|V|$. This is because of the vertices that are, at the same time, center-vertices and part of other sub-graphs, such as vertex B in Figure 1.

The multiset of vertices in the distributed graph $V^D = \biguplus_{v \in C} V_v$ increases linearly with the number of edges in each sub-graph:

$$|V^D| = \sum_{v \in C} |V_v| = \sum_{v \in C} (|E_v| + 1)$$

If the number of outgoing edges for each vertex $v \in C$ is limited by a constant K, then $|E_v| + 1$ can be substituted with $K + 1$ and the overhead is constant:

$$\frac{|V^D|}{|V|} \le \frac{\sum_{v \in C}(K+1)}{|V|} = \frac{|C| \cdot (K+1)}{|V|}$$

In the worst case, all vertices in V have outgoing edges. Hence, C is equal to V and thus

$$\frac{|V^D|}{|V|} \le \frac{|V| \cdot (K+1)}{|V|} = K + 1$$

Note, that the space complexity of the distributed graph is $O(|V|)$, independent of how many physical nodes are used.

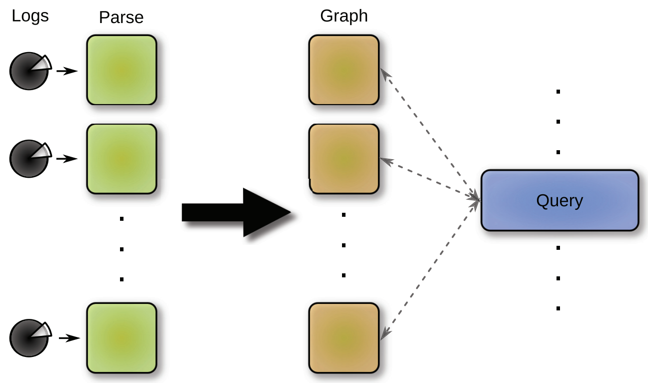

Figure 2: Distributed graph architecture

Our algorithm maintains two graphs for every center vertex. One contains the information of all the data received within a given window (e.g., the last 100 updates for each center-vertex) and the other one stores the summary information of the complete data that has been received so far. The latter uses a top-k algorithm and the former its window size to limit the number of outgoing connections per vertex to K. We show an example of how to use and merge both graphs in Section 3. Both graphs are bounded in size. Therefore, the amount of data that is stored per node is bounded by the size of the two graphs times the number of center vertices, stored on that particular node. An algorithm for reconstructing the original graph G is given in Section 2.4.

2.2 Processing architecture

Figure 2 depicts the architecture of the distributed graph processing. The log data can be read from any socket. This includes disks, as well as TCP-connections and Unix-pipes. The *parse component* is used to extract and convert the necessary information from the log data. For example, the parse component can be used to construct a numerical representation of a human-readable IP address, such as “141.30.2.2”.

Subsequently, the parse component generates a new event and sends it downstream to the *graph components*. All communication in our system is event based. Events are key-value pairs. The hash of the key of an event is used to route it to the responsible downstream node. For example, if there are two graph components, then all events with even hashes of their key will be routed to the first, and all events with odd hashes of their key will be routed to the second graph component. The processing is also coupled to events and their keys. If an event with key k is received, only the sub-graphs whose center vertex corresponds to k will be read or written. Thus, events with different keys can be processed in parallel, using multiple threads.

Considering the example from Figure 1, if we want to send an event to update the sub graph with the center vertex B, our event has to have the key B. We will show how we chose the keys for our two example applications in Section 3.

Since the system uses back pressure, the processing speed of each component is not only limited by itself but also by all downstream components. For example, the parse components cannot send more events than the graph components can process.

2.3 Simple queries

The graph maintained by the graph components can be accessed and queried. There is no limit on the number of queries running simultaneously. Moreover, each query itself may be distributed using the same concepts as the distributed graph for communication and parallelization. Since the routing scheme is static, it is straight forward to compute where the graph of a specific item can be found.

The query interface provided by the graph components is based on events. So called *control messages* can be sent to request the graph of a given item and will be answered with a *response message*. Algorithm 1 shows how a query is performed.

Algorithm 1: Query a single sub-graph

```
input  : vertex
output: sub-graph
begin function query
    index = hash(vertex) % ngraph_nodes;
    control_message m;
    m.vertex = vertex;
    // emits the event and waits for the answer
    sub_graph = emit_event(index, m);
    return sub_graph
end
```

2.4 Multi-level queries (transitive closures)

To obtain the transitive closure of a sub-graph or even the complete graph, the previous algorithm has to be repeated recursively. Algorithm 2 shows this in pseudo-code. The algorithm takes the initial vertex and the desired depth as inputs. Initially, the set *visited* and the graph are initialized to empty sets. First, we check if the sub-graph of a vertex has already been processed. This ensures each vertex is included only once in the output. Thereafter, we check for each adjacent vertex in the sub-graph if it has already been processed, and if the remaining search depth is still greater than 0, we request the sub-graph of the current adjacent vertex. The result is the union of all the collected subgraphs.

3. EXAMPLES

In the following, we will provide two use-cases that we have investigated using our approach. The first is a fraud detection application for the telephony domain and the second is an attack detection application for the internet domain. Both examples use community of interest graphs (COI) [7]. Algorithm 3 shows how the COI graph is constructed. We first add the received entry to the window, which is the first graph we store for every center vertex. If more than a predefined number of connections have already been added to the window, the window is merged with the (potentially not yet) existing COI, which is the second graph we store for every center vertex (topk). To this end, the application needs to define an attribute by which to measure the weight of the connections. With that, the weight (sum of all the attribute values) in the window, multiplied with a damping factor $1 - \theta$, are added to the weight (multiplied with θ) in the COI. Since $\theta = 0.85$ in both examples, the influence of the new connections in the window is damped. Thereafter, the weights of all contacts in the COI that have not been observed during the current window are decayed by multiplying them with θ. To keep the COI at a maximum size of K, we remove the weakest link until the size of the COI is smaller or equal to K. Finally, the window and the counter are reset.

Algorithm 2: Transitive closure of a COI.

```
global visited = ∅;
global graph = ∅;
input  : vertex, depth
output: graph
begin function closure
    if vertex ∈ visited then
        return ∅
    end
    visited.insert(vertex);
    // the query function is shown in algorithm 1
    sub_graph = query(vertex);
    graph = graph ∪ sub_graph;
    if depth > 0 then
        foreach adjacent_vertex ∈ sub_graph do
            if adjacent_vertex ∈ visited then
                continue
            end
            sub_graph = closure(adjacent_vertex, depth - 1);
            graph = graph ∪ sub_graph;
        end
    end
    return graph
end
```

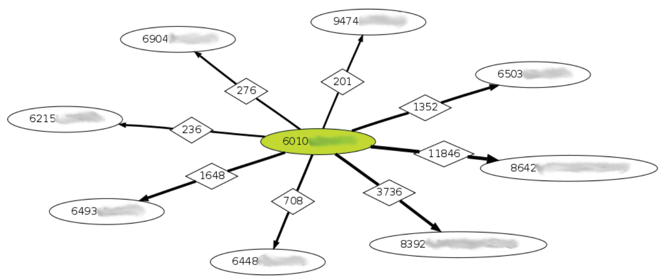

Figure 3: Community of interest example for an anonymized phone number

3.1 Telephony application

In this application, we parse *call-detail-records* (CDRs) that contain various information for each telephone call. The CDRs are stored in a text file with each line representing one CDR. The parse component extracts the caller, callee, and call-duration information from each CDR. Thereafter, two events are sent to the graph component: one with the caller's hash as the key and one with the callee's hash as the key. Each event message is about 24 bytes. In the graph component, we construct a sub-graph for each event's key, i.e., there is a sub-graph for each phone number observed in the CDRs—be it the caller or the callee. This approach is similar to [7], with the exception that we dynamically update each sub-graph individually, instead of updating the complete graph after a fixed time interval.

3.1.1 Graph

Figure 3 shows the top-k sub-graph of a phone number.

Algorithm 3: Top-k graph construction

```
input  : (window, topk, counter, new_contact, attribute)
output: None
begin
    // Save new contact in window
    window[new_contact].weight += attribute;
    counter++;
    // Merge window into top-k after $N$ events
    if $counter > N$ then
        foreach $contact \in window$ do
            // $\theta$ has a value of 0.85 in our analysis.
            ww = window[contact].weight;
            tw = topk[contact].weight;
            topk[contact].weight = 1 - $\theta$ * ww + $\theta$ * tw;
        end
        // Decay weight of old connections
        foreach $contact \in topk \wedge contact \notin window$ do
            topk[contact].weight = $\theta$ * topk[contact].weight;
        end
        // Copy additional information $I$ which can be
           saved with the edges
        // Remove the weakest links
        while $|topk| > K$ do
            remove_weakest_link_from(topk);
        end
        window = $\emptyset$;
        counter = 0;
    end
end
```

The adjacent vertices are other phone numbers that have either been callers or callees with respect to communication with the center vertex. Edges are weighted by the sum of all call durations of each communication between two vertices. The graph in the figure was obtained using the simple query function (see Algorithm 1). All links were stored into a file and then rendered using the *neato* tool, contained in the *graphviz* [10] package.

3.1.2 Social fingerprinting

One application of communities of interest is social fingerprinting. The idea is that an individual can be identified, with a certain probability, using only its community of interest. The probability depends highly on the size of the COI. Research has shown that a COI of size 9 is sufficient [7]. This, in turn, can be used to check whether the same individual lies behind two or more different numbers. For example, if we have a set of known fraudsters, we can check if the COI of a suspicious customer is sufficiently similar (e.g., 90%) to one of the stored fraudster's COIs. This check needs to be done in real time, since action must be taken (e.g., block a call) before the actual call is completed.

Implementing such a fingerprint query on top of our graph-mining system is straight forward and shown in Algorithm 4. First, we query the COI of the number in question. Then we construct the set intersection for the received COI with each of the stored fraudster's COIs. If at least one of the intersections is as large as 90% of the COIs size, a possible fraudster has been found and a fraud analyst should further investigate the case.

3.2 Netflow application

In this application, we parse netflow logs, recorded by routers, which contain various information about internet

Algorithm 4: Fingerprinting

```
input  : suspect_number
output: alarm
begin
    suspect_coi = query(suspect_number);
    foreach $coi \in fraudster\_cois$ do
        intersection = coi $\cap$ suspect_coi;
        if $|intersection| >= 0.9 * |coi|$ then
            return $true$
        else
            return $false$
        end
    end
end
```

based communication connections. Each netflow entry is a line in a text file. We use the parse component to extract the source-ip, target-ip, source-port, target-port and number of transferred bytes from each netflow entry. Thereafter, an event, representing one netflow entry, is sent to the filter component. Each event message is about 32 bytes.

The filter component is a new component, introduced for the netflow application, to discard unimportant traffic before it reaches the graph component. The decision to discarded an event depends on various factors, such as the ports used and source and destination IPs. A detailed description of the filter component is out of the scope of this paper, but fundamentally we use this component to filter traffic that is not related to the community or uses trusted protocols. Finally, if the original event was not filtered, the filter sends an event to the graph components using the source-ip's hash as the key.

With that, we construct a sub-graph for each unfiltered IP address. We use the number of bytes transferred to determine the weight of each edge and also store the used ports as additional information for each edge.

3.2.1 Graph

Figure 4 shows a top-k sub-graph as it is used in the internet attack detection application. The center vertex depicts an IP address that connects to various members of the community "A" and three members of the community "B". We omit the real community names for privacy reasons. The edges depict communication between the two vertices and the diamond boxes depict the weights of the connections. These weights are determined by the sum of the transferred bytes by each communication between two vertices. While not shown in the figure, the edges also contain every source- and destination-port combination ever used by the two vertices. The graph in the figure was obtained using the simple query function (see Algorithm 1).

3.2.2 Security incident detection

The netflow application is used to detect stealthy security incidents by utilizing information across a community of organizations (e.g., banking industry, energy generation and distribution industry). A stealthy attack is, for example, an attack which only targets one or few machines in each organization and does not transfer large volumes of information at once. Consequently, it may not be detected by any standard mechanisms that checks in- and out-going traffic for unusual behavior.

The basic idea of our community-based security incidence

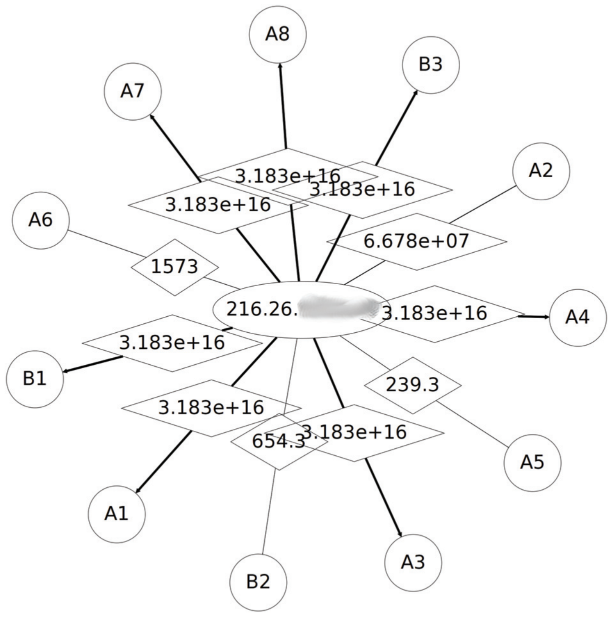

Figure 4: Community of interest example for an anonymized IP address

is to identify IP addresses outside the community that communicate with a number of different organizations in the community. Typical examples of security threats that can be detected using this approach include botnet controllers managing a number of bots in the community, compromised machines downloading stolen information to a dedicated server, an attacker targeting machines in multiple organizations, as well as many security policy violations (e.g., illegal software download sites, etc). Thresholds can be used to control the number of alarms generated.

Algorithm 5: Internet attack detection query

```
input  : topk, community, threshold
output: alarm
begin
    count = 0;
    source-ip = get_center_vertex(topk);
    foreach contact ∈ topk do
        if contact ∈ community then
            ++count;
        end
    end
    if count > threshold then
        if source-ip ∈ community AND used ports are not
        suspicious then
            return ∅
        else
            return source-ip
        end
    else
        return ∅
    end
end
```

To determine if an IP address tries to attack the community, Algorithm 5 is executed every time a new entry in the netflow is forwarded to the graph component. The algorithm receives the top-k graph, together with the community to check and a threshold. It then iterates over all contacts in the top-k graph and counts with how many community members the corresponding center vertex has contact. If this number is higher than the threshold and the center vertex is not in the community itself or uses suspicious ports, an alarm is raised. A detailed discussion of what is considered to be a suspicious port is out of the scope of this paper. The generated alarms are forwarded to an external service, e.g. a web-server.

4. EVALUATION

We executed all benchmarks of the telephony application on a 50 node cluster, each equipped with two Intel Xeon quad-core processors and 8GB of RAM. The relation of nodes for this experiment is $(n : n+1)$: n parse nodes and $n+1$ graph nodes.

The experiments, using the netflow application were executed on a 9-node cluster, each equipped with 4 Intel Xeon quad-core processors and 24GB of RAM. The relation of nodes for this experiment is $(n : n : 1)$: n parse nodes, n filter nodes, and 1 graph node. Each node is a physical machine.

The measurements were conducted by processing accumulated (i.e., historic) data, because the volume of the real time data was not high enough (for example, we can process one day of netflow data in 40 minutes on a single machine) to show how the system behaves (i.e., with regard to latency and throughput) at its limit. Of course, we plan to integrate our system with real-time analysis at a later point in time.

4.1 Telephony application

Figure 5 shows the scalability of the dynamic graphs. It can be seen that the dynamic graphs scale linearly with the number of nodes used and, more importantly, with the input traffic. The linear characteristic is due to the fact that we use the same set of CDRs for every setup. Thus, the number of sub-graphs per node decreases as more nodes are added. In the largest configuration, we can process up to 39 million log-entries per second.

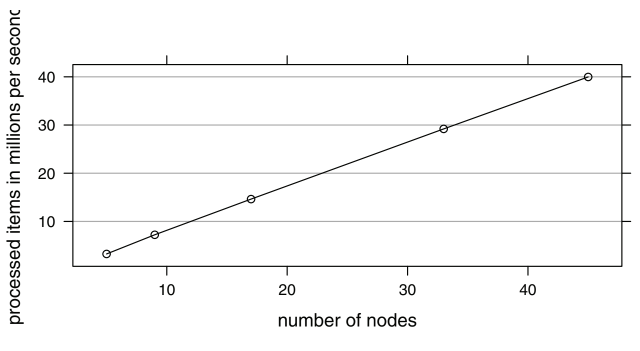

Figure 5: Scalability of dynamic graphs

This is still being done in near real time: Figure 6 shows the mean latency from reading a new log entry until the processing is completely finished for different configurations. It can be observed that by adding more nodes, the overall latency declines. The reason for this is the same as for the linear scalability in the throughput measurement. Note, however, that the throughput also increases with larger setups (as shown in Figure 5). Therefore, by increasing the number of nodes, latency can be maintained at a constant level even if the workload increases.

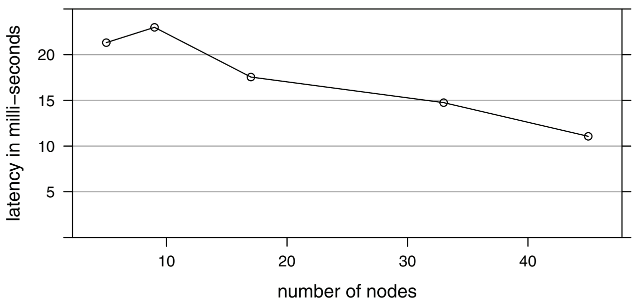

Figure 6: Latency of processing individual log entries

One of the motivations for mining log data with a distributed system is that the computational overhead can be distributed. For example, the nodes that parse the entries from the log files will not have to dedicate all their resources to the mining and may be used for other purposes. This argument is supported by Figure 7. It shows that the CPU of the parse components is much less utilized than that of the graph components. However, the network connections of the parse components are fully utilized (i.e., 115MB/s) because the parsed entries need to be sent to the graph components.

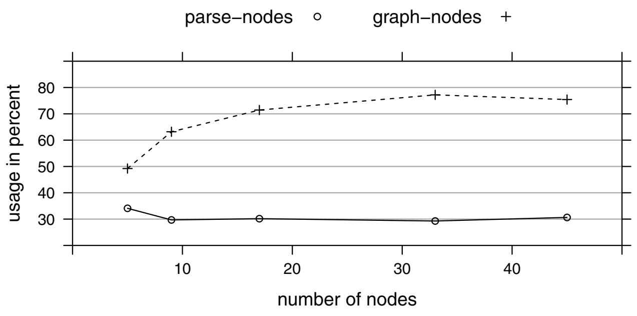

Figure 7: CPU utilization, separated for source and graph nodes

It can also be seen that larger configurations yield a higher CPU utilization on the nodes running the graph components. This is caused by the fact that we always use 1 graph component more than parse components ($n : n + 1$), which is the setup in which the telephony application performs optimally. Since this is constant, its influence is higher for smaller configurations and the graph nodes in smaller configurations will be less utilized.

4.2 Netflow application

We executed the same set of measurements for the netflow application. However, the results cannot be compared directly since the hardware is different[1] and the netflow application uses one additional component (the filter).

[1]Due to legal reasons we were not able to execute all experiments on the same infrastructure.

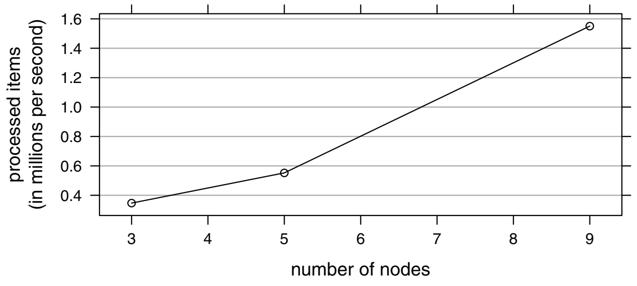

Figure 8: Scalability of dynamic graphs

Figure 8 shows that the netflow application scales almost linearly with the number of nodes. Figure 9 depicts the mean latency from the time a netflow entry is read until the graph component finished processing it completely. As expected, the latency is significantly higher than for the telephony application. This is due to (1) a non-optimal implementation of the filter and (2) the network delay, introduced by the additional filter component. The reason that the latency decreases for larger configurations is that the filter components, which dominate the overall latency, are less loaded.

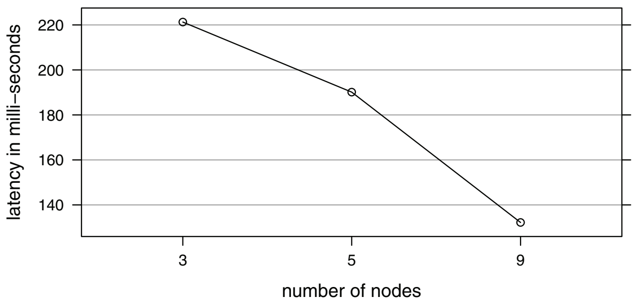

Figure 9: Latency of individual log-entries

The CPU utilization (see Figure 10) shows that the filter components have the highest CPU utilization. The parse components do not require as much CPU since they are limited by I/O operations (reading the log entries from disk and sending events to the filter components). Again, the reason for the decrease of CPU utilization in larger configurations is mostly due to lower load on the filters.

Interestingly, however, the graph component is much less utilized than for the telephony application. This is true for the CPU utilization (see Figure 10) as well as the network traffic. As depicted in Figure 11, the network throughput from the parse component to the filter component (the graph on the left side) is by orders of magnitude higher than the network throughput from the filter component to the graph component (the graph on the right side). Therefore, also the CPU utilization of the graph component is by orders of magnitude lower than for the telephony application, where the graph component has to process all the traffic sent from the parse component.

Consequently, CPU utilization can be traded for network traffic: Either the CPU usage is lower but the network traf-

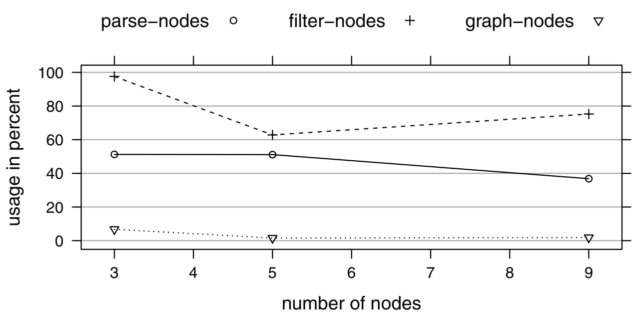

Figure 10: CPU utilization, separated for sources, filter and graph nodes

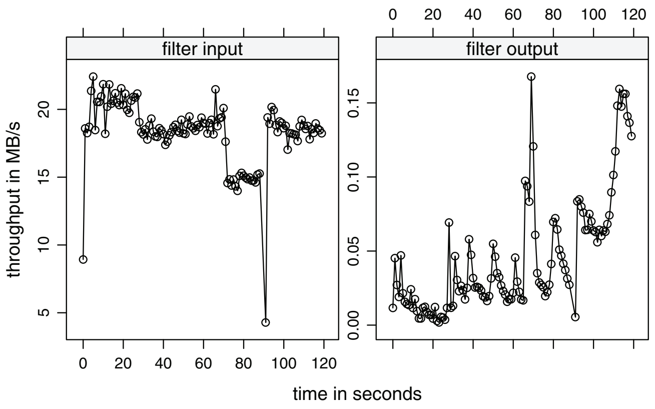

Figure 11: Influence of filtering on throughput

fic is higher (as shown in Figure 7) or the CPU usage is higher but the network utilization is lower (as shown in Figures 10 and 11).

5. RELATED WORK

The related work can be split into two distinct categories: work in data mining and work on graphs in general. The latter includes distributed graphs as well as fraud detection with graphs.

5.1 Data mining

A system close to our mining platform is IBM's SPC [2]. It provides a rich API for defining operators. However, it only reaches high throughput with large event sizes (more than $128KB$) and the processing elements can only operate on fixed windows over the input stream, while our system allows a state to be stored independently from a window (such as the top-k graphs).

Dryad [13] is a streaming system by Microsoft. It relies on the availability of a distributed file system and processing elements are executed in a single-threaded manner. No latency measurements or absolute numbers on the achieved throughput are provided in [13]. SCOPE [5] is a system by Microsoft, specifically designed to parse log files. It relies, like Dryad, on the availability of a distributed file system. Furthermore, it provides only limited customizability of operators, i.e., operators need to be defined by means of the three constructs: PROCESS, REDUCE, and COMBINE. No absolute latency or throughput results are provided in [5] and the scalability is sub-linear.

Hadoop [1] is the main open-source implementation of the MapReduce paradigm [8]. It is intended for batch processing and therefore provides latencies far beyond the seconds range. Several variations of the model have also been proposed. Hadoop Online [6] improves Hadoop's efficiency and latency by enabling a direct communication between mappers and reducers. In the original publication, the authors consider the processing of continuous data streams, but provide only a minimal example and no performance evaluation. State is incorporated into MapReduce in [15]. However, the goal is to provide better support for incremental batch jobs and, thus, it will not support applications requiring low latency. Better incremental support for batch systems is also the focus of the new Google system, named Percolator [16]. Unfortunately, the observed latencies in Percolator can range up to minutes.

Borealis [18] addresses scalability by optimizing the placement of operators, balancing load, and applying sophisticated techniques for load shedding. However, it only supports SQL-like operators. Moreover, it does not address the problem of parallelizing a single operation among a large number of nodes. GSDM [14] addresses operations that are partitionable, like is the case with MapReduce, but it does not consider operators that maintain state—operator process whole windows at a time, similar to our jumping windows.

Finally, StreamMine [3, 4] addresses low latency, fault tolerance, and parallelization of stateful operators, but the speculative approach does not scale horizontally (i.e., among nodes within a cluster).

5.2 Graphs

The use of graph structures for different kinds of applications has been addressed in number of research efforts [21, 11, 9, 19]. However, there is very little related work on how to efficiently distribute huge graphs. The authors in [12] propose a distributed graph algorithm but focus only on finding the strongly connected components. It is not clear if the graphs can be updated and with what latencies. Taentzer [17] provides a theoretical analysis of distributed graphs, but no implementation is provided.

In the field of community of interest-based fraud detection, the closest related work is [7]. However, the authors do not consider constant updates but rather batch processing. Moreover, the computation and the graph are not distributed.

The indexing of graph structures in databases has been investigated in [20]. Nevertheless, the analysis considers only one computer (not distributed), and latencies for queries and updates are in the range of in the seconds with larger databases.

6. CONCLUSIONS

We have demonstrated how to use graphs for many interesting log processing problems. The evaluation showed that we can process huge amounts of log data in near real time (i.e., 10 ms). This is especially due to the very good scalability of our system. Since the index is deterministic and easy to compute, queries are very simple to implement. Addition of new processing phases, such as our filter components, is easy and can result in significant reduction of data transferred and processed. Our future work includes evaluation of our approach in new application domains and analysis problems, including design and implementation of additional processing components.

Acknowledgment

This research was funded as part of the SRT-15 project supported by the European Commission under the Seventh Framework Program (FP7) with grant agreement number 257843.

7. REFERENCES

[1] Hadoop. `http://hadoop.apache.org/`, January 2010.

[2] L. Amini, H. Andrade, R. Bhagwan, F. Eskesen, R. King, P. Selo, Y. Park, and C. Venkatramani. Spc: a distributed, scalable platform for data mining. In *Proceedings of the 4th international workshop on Data mining standards, services and platforms*, DMSSP '06, pages 27–37, New York, NY, USA, 2006. ACM.

[3] A. Brito, C. Fetzer, and P. Felber. Minimizing Latency in Fault-Tolerant Distributed Stream Processing Systems. In *ICDCS '09: Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems*, pages 173–182, Washington, DC, USA, 2009. IEEE Computer Society.

[4] A. Brito, C. Fetzer, H. Sturzrehm, and P. Felber. Speculative out-of-order event processing with software transaction memory. In *DEBS '08: Proceedings of the second international conference on Distributed event-based systems*, pages 265–275, New York, NY, USA, 2008. ACM.

[5] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.*, 1:1265–1276, August 2008.

[6] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. In *NSDI'10: Proceedings of the 7th USENIX conference on Networked systems design and implementation*, pages 21–21, Berkeley, CA, USA, 2010. USENIX Association.

[7] C. Cortes, D. Pregibon, and C. Volinsky. Communities of interest. In F. Hoffmann, D. Hand, N. Adams, D. Fisher, and G. Guimaraes, editors, *Advances in Intelligent Data Analysis*, volume 2189 of *Lecture Notes in Computer Science*, pages 105–114. Springer Berlin / Heidelberg, 2001. 10.1007/3-540-44816-0_11.

[8] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[9] D. Donato. Graph structures and algorithms for query-log analysis. In F. Ferreira, B. Löwe, E. Mayordomo, and L. Mendes Gomes, editors, *Programs, Proofs, Processes*, volume 6158 of *Lecture Notes in Computer Science*, pages 126–131. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-13962-8_14.

[10] J. Ellson, E. Gansner, L. Koutsofios, S. North, and G. Woodhull. Graphviz–open source graph drawing tools. In P. Mutzel, M. Jünger, and S. Leipert, editors, *Graph Drawing*, volume 2265 of *Lecture Notes in Computer Science*, pages 594–597. Springer Berlin / Heidelberg, 2002. 10.1007/3-540-45848-4_57.

[11] A. Francisco, R. Baeza-Yates, and A. Oliveira. Clique analysis of query log graphs. In A. Amir, A. Turpin, and A. Moffat, editors, *String Processing and Information Retrieval*, volume 5280 of *Lecture Notes in Computer Science*, pages 188–199. Springer Berlin / Heidelberg, 2009. 10.1007/978-3-540-89097-3_19.

[12] W. M. III, B. Hendrickson, S. J. Plimpton, and L. Rauchwerger. Finding strongly connected components in distributed graphs. *Journal of Parallel and Distributed Computing*, 65(8):901 – 910, 2005.

[13] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM.

[14] M. Ivanova and T. Risch. Customizable parallel execution of scientific stream queries. In *Very Large Data Bases*, pages 157—-168. VLDB Endowment, 2005.

[15] D. Logothetis, C. Olston, B. Reed, K. C. Webb, and K. Yocum. Stateful bulk processing for incremental analytics. In *SoCC '10: Proceedings of the 1st ACM symposium on Cloud computing*, pages 51–62, New York, NY, USA, 2010. ACM.

[16] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, Berkeley, CA, USA, 2010. USENIX Association.

[17] G. Taentzer. Distributed graphs and graph transformation. *Applied Categorical Structures*, 7:431–462, 1999. 10.1023/A:1008683005045.

[18] N. Tatbul, U. Çetintemel, and S. Zdonik. Staying fit: efficient load shedding techniques for distributed stream processing. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 159–170. VLDB Endowment, 2007.

[19] P. Verkaik, O. Spatscheck, J. Van der Merwe, and A. C. Snoeren. Primed: community-of-interest-based ddos mitigation. In *Proceedings of the 2006 SIGCOMM workshop on Large-scale attack defense*, LSAD '06, pages 147–154, New York, NY, USA, 2006. ACM.

[20] D. Williams, J. Huan, and W. Wang. Graph database indexing using structured graph decomposition. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 976 –985, april 2007.

[21] A. K. C. Wong and M. You. Entropy and distance of random graphs with application to structural pattern recognition. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, PAMI-7(5):599 –609, sept. 1985.