Grand Challenge: GraphCEP - Real-time Data Analytics Using Parallel Complex Event and Graph Processing

Ruben Mayer, Christian Mayer, Muhammad Adnan Tariq, Kurt Rothermel
IPVS, University of Stuttgart, Germany
{first name.last name}@ipvs.uni-stuttgart.de

ABSTRACT

In recent years, the proliferation of highly dynamic graph-structured data streams fueled the demand for real-time data analytics. For instance, detecting recent trends in social networks enables new applications in areas such as disaster detection, business analytics or health-care. Parallel Complex Event Processing has evolved as the paradigm of choice to analyze data streams in a timely manner, where the incoming data streams are split and processed independently by parallel operator instances. However, the degree of parallelism is limited by the feasibility of splitting the data streams into independent parts such that correctness of event processing is still ensured. In this paper, we overcome this limitation for graph-structured data by further parallelizing individual operator instances using modern graph processing systems. These systems partition the graph data and execute graph algorithms in a highly parallel fashion, for instance using cloud resources. To this end, we propose a novel graph-based Complex Event Processing system GraphCEP and evaluate its performance in the setting of two case studies from the DEBS Grand Challenge 2016.

CCS Concepts

•**Applied computing** → **Event-driven architectures;**
•**Computer systems organization** → *Cloud computing;*

Keywords

Complex event processing; distributed graph processing

1. INTRODUCTION

An increasing amount of streaming data from various sources accrues in modern IT systems, stemming, e.g., from mobile devices and connected sensors. These data streams often exhibit a graph structure, i.e., elements from the different data streams are related to each other. For instance, in an evolving social network, commments refer to existing posts and users, which themselves are connected via friendship relations. As the available processing capabilites increase by technological progress such as multi-core computers and Cloud Computing, the demand for processing the data streams arises in order to enable new applications that react on situations of interest in the surrounding world in near real-time. Such situations can be, e.g., evolving trends in social networks or the detection of strong communities that are interested in a similar topic. They can be detected from patterns occurring in the incoming data streams.

For detecting patterns in data streams, the paradigm of parallel Complex Event Processing (CEP) has been established in the past decade [17]. Parallel CEP systems are nowadays widely used in many different application areas, such as algorithmic trading and health care [1]. In doing so, state-of-the-art CEP systems are able to detect multiple instances of a queried pattern on the incoming streams in parallel by splitting the streams into partitions that are processed in parallel by an elastic number of operator instances. To be able to detect all occurring patterns, operator instances need to receive and process complete partitions, i.e., a partition can not be further divided, but has to be processed completely by one operator instance. However, due to the ever-increasing data rates and complexity of patterns, even detecting a single instance of a pattern already becomes a bottleneck in terms of CPU and memory requirements. Partitions become larger, which results in an accumulation of a large processing state in the operator instances. This becomes especially evident in the context of analytics on graph-structured data, where billions of edges and millions of vertices may need to be analyzed in order to detect a queried pattern. For instance, finding time-varying communities of friends interested in similar topics requires to execute graph-theoretical algorithms on friendship graphs.

On the other hand, in the recent past, graph processing systems have seen an enormous boost (Pregel [14], GrapH [15]). These systems are capable of performing data analytics on large graphs in parallel, which is exactly what is lacking in parallel CEP systems. However, most of these systems work in a batch-like execution mode: they load graph data from files, perform graph algorithms on it and store the results on files. Hence, they lack support for efficiently processing streaming data.

In this paper, we pose the question whether parallel CEP and graph processing can be combined in order to solve the challenges of streaming graph-structured data analysis. Our answer is GraphCEP, an integrated system that combines the strengths of parallel CEP systems with the strengths

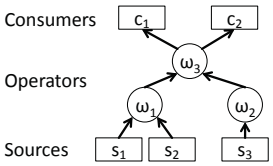


Figure 1: Complex Event Processing.

of parallel graph processing systems. Toward this end, we make the following contributions: (1) We propose an architecture combining parallel CEP and graph processing—called GraphCEP—, and (2) we show the benefits of the proposed architecture in the course of two case studies from the DEBS Grand Challenge 2016.

2. CHALLENGES AND GOAL

Complex Event Processing (CEP) is a paradigm that is nowadays widely used in order to detect situations of interest in the sourrounding world in situation-aware applications. Those situations correspond to patterns in source event streams (e.g., stemming from sensors or from other applications such as social networking applications). In doing so, the patterns are specified by domain experts in CEP query languages such as TESLA [5] and Cordies [11].

According to the queries, the CEP system deploys a distributed *operator graph* $G(\Omega \cup S \cup C, L)$ interconnecting sources in S, operators in Ω, and consumers, i.e., applications interested in queried patterns, in C in form of event streams in $L \subset (S \cup \Omega) \times (\Omega \cup C)$ (cf. Fig. 1) [12]. An operator $\omega \in \Omega$ detects patterns on its incoming event streams in so-called correlation steps. In each correlation step, ω works on a *selection* σ which is a finite subset of events from the incoming streams. A correlation function $f_\omega : \sigma \rightarrow (e_1, \ldots, e_m)$ specifies the mapping from the processed selection to a set of complex events emitted by the operator, which resemble pattern matches on the incoming event streams.

When single operators in a distributed CEP system are exposed to a heavy workload, the processing capabilities of a sequentially executed operator are often not sufficient. Hence, parallelization of CEP operators has emerged as an important research topic in the recent past. Generally, there are two ways of parallelizing a CEP operator: Intra-operator parallelization and data parallelization. In intra-operator parallelization [1], also known as pipelining, internal processing steps of an operator are derived from the query and executed in parallel on different CPU cores or processing nodes. This approach offers only a limited achievable parallelization degree, as the number of steps in detecting a queried pattern is limited by the number of variables in the query, and hence, is operator-specific. A more promising approach is data parallelization [17], where the incoming event streams of an operator are split into partitions that can be processed independently by a set of identical copies of the operator, called operator instances, that run in parallel.

However, graph-structured streaming data poses a set of challenges on data parallelization in CEP. In the following section, we analyze and discuss the challenges that arise.

2.1 Challenges

Graph-structured streaming data poses several challenges which current CEP parallelization frameworks do not properly address. The first challenge is centered around consistent splitting of event streams, such that no false-negatives and no false-positives are produced. The second challenge deals with the granularity of parallelization of graph-based data. The third challenge regards the streaming nature of the graph-based data.

Consistent Splitting. The challenge in splitting the incoming data streams is to find partitions that contain exactly those events needed in order to detect the queried patterns in the operator instances. On the one hand, this means that each operator instance receives exactly those events that lead to the detection of patterns in the original incoming data streams, such that all existing patterns are actually detected (no false-negative detections). On the other hand, the operator instances shall not detect patterns that are not present in the original data streams (no false-positives). Consistent splitting is reached when the partitions contain exactly those events that are in the selections of an operator w.r.t. its correlation function.

Both false-negatives and false-positives can occur when the splitting function assigns events from one selection partly to one operator instance and partly to another operator instance. Then, none of the instances has the complete view of an occurring pattern, and hence, the output is not consistent with a sequential execution where the operator would have the complete view. For instance, in a CEP operator that detects a community of people who like the same comments in a social network, all likes of the same comment have to be assigned to the same operator instance. If this was not the case, only parts of the community could be detected and the result would be inconsistent.

Granularity of Parallelization. Apart from the challenge of splitting the incoming data streams, the concept of data parallelization itself is limited to the parallelism inherent to the data. That means, that an event selection has to be processed completely by one operator instance. If the selection contains a large graph, it cannot be split further into independent parts. This limits the scalability of the system, as large graphs are very common in graph-structured data. For instance, in the aforementioned operator that detects a community of people who like the same comments, a comment with a large number of likes has to be completely analyzed by one single operator instance.

Streaming Graph-Structured Data. In addition to splitting the data streams and providing a high degree of parallelization, the CEP system also has to efficiently exploit the streaming nature of the graph structured data. This implies that the graph structure is evolving in an incremental fashion, for instance, when friendship relations are built or destroyed in a social graph at different points in time. This is in contrast to static graph-structured data sets, where a graph does not evolve over time, but can be completely imported into the system at startup and then is only queried.

2.2 Goal

Our goal is to devise a distributed, parallel CEP system that is able to efficiently and scalably process streaming graph data. To achieve this goal, the system has to be capable of finding consistent stream splitting functions, such that no false-negative and no false-positive pattern detections are produced. Further, it has to be able to parallelize the processing of graph data beyond the capabilities of state-of-the-art parallel CEP frameworks that only split the incoming data streams. Finally, the system shall be able to handle streaming graph data in an efficient manner, that is, in a

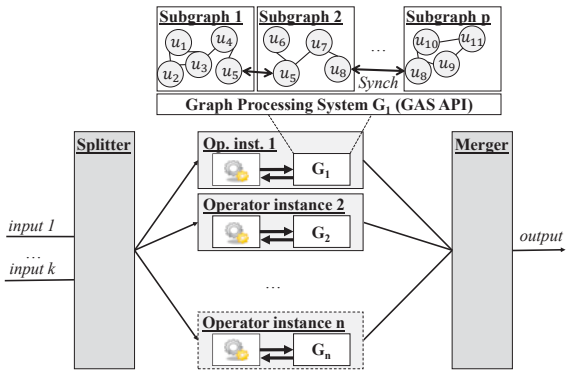


Figure 2: Architecture GraphCEP

stream-aware processing model in contrast to a batch-like processing model.

We believe, that in order to devise such a system, it is necessary to unify research and developments from two areas, Complex Event Processing and Graph Processing. In the following, we describe our unified framework GraphCEP.

3. GraphCEP

We propose GraphCEP, a framework that combines state-of-the-art stream partitioning technology in the splitter with distributed graph processing in the operator instances.

GraphCEP consists of a split–process–merge architecture (cf. Fig. 2). The splitter is in charge of finding consistent splitting points in the incoming event streams, such that operator instances do not miss detecting any occurring pattern instances (no false-negatives), but also not erroneously detect pattern instances that are not in the incoming event streams (no false-positives). The operator instances run in parallel and use an interface to a full-fledged distributed graph processing system, such that the processing of graph-structured data in single operator instances can be further parallelized. Finally, a merger reorders the concurrently detected events from the operator instance into a deterministic order. In the following, we describe the single components in more detail.

Splitter. Event streams are split into partitions by two logical predicates, $P_s : e \rightarrow \texttt{BOOL}$ and $P_c : (partition, e) \rightarrow \texttt{BOOL}$. P_s evaluates to true when an event opens a new partition, whereas P_c evaluates to true when an event closes an open partition. The predicates are programmed by a domain expert. Using the predicates, GraphCEP is able to split incoming event streams for all different kinds of operators, such as operators working on time-based or tuple-based sliding windows, sequence operators, OR operators, AND operators, and aperiodic and periodic operators (cf. [17]).

Additionally to the predicate-based splitting, GraphCEP can also be deployed with a *key-based* stream partitioning mode. In this mode, events are assigned to operator instances according to a key value they contain. This can be, for instance, a user id, a location, a stock symbol, etc. In doing so, specific value ranges are assigned to specific operator instances. In the key-based partitioning mode, the domain expert needs to specify the assignment of value ranges to operator instances, e.g., which user ids, locations or stock symbols are assigned to which operator instance.

Operator Instance. Operator Instances detect patterns on the incoming event streams and emit events to the merger. They are programmed by domain experts according to the patterns to be detected. The specialty of GraphCEP is that the domain expert has access to a fully-fledged distributed graph processing engine. This access is given via two interface functions: `manipulate_graph`() and `execute`(). The former interface function `manipulate_graph`() is for adding and remove vertices and edges from the graph model in the graph processing engine. The operator instance decides, based on the queried patterns, which events lead to such a graph manipulation. For instance, it can decide that a like of a comment leads to adding a node for the like and an edge from that like to the liked comment. Another example is that the timeout of a comment leads to the removal of nodes and edges. The latter interface function `execute`() triggers the execution of a graph-theoretic algorithm—such as, e.g., `Max_Clique`—which is then performed on the existing graph model in a highly-parallel, distributed fashion.

The graph processing algorithms in the graph processing engine are programmed using the well-known Gather–Apply–Scatter (GAS) paradigm [8]. Here, the graph algorithm programmer specifies a *vertex function* to be executed by each vertex in the graph. For instance, vertex v collects data from its neighbors in the gather phase, changes its own vertex data in the apply phase using the gathered neighboring data, and initiates execution of vertex functions on its neighbors, if v's data has changed.

The proposed architecture allows for a separation of concerns between the CEP and the graph processing domain: The CEP domain expert does not have to care about parallelization of graph processing, but can just execute the graph-theoretic algorithms available in the graph processing engine via the `execute`() function. On the other hand, the graph processing domain expert does not have to deal with pattern matching in CEP, but can focus on efficient parallel graph processing algorithms. This way, GraphCEP combines the best of both worlds: CEP pattern matching functionality and graph processing efficiency and scalability.

Merger. The merger component brings the emitted events from the operator instance into a deterministic order. This order can be, for instance, based on timestamps of events in a pattern. A domain expert can specify the ordering conditions, and further specify combination functions, e.g., to emit top-k-lists.

4. CASE STUDIES

In this section, we show how to apply GraphCEP to real-world data analytics using two case studies in the context of evolving social network graphs, posed by the DEBS Grand Challenge 2016. The first case study is identification of the posts that currently trigger the most activity in the social network. The second case study deals with the identification of large communities that are currently involved in a topic. A detailed description of the case studies is given in [9].

4.1 Case Study 1: Posting Activity Detection

The first case study deals with detecting posts that currently trigger the most activity in a social network. The goal of the query in this scenario is to compute the emerging top-k list of most active posts.

The activity of a post is measured by means of a point system. At creation, a post gets a point score of 10. Further, for each comment to a post, the commented post gets additional 10 points. However, as time passes, the points time out: each 24 hours since its creation, a post loses 1

point and each 24 hours since creation of a comment to a post, the commented post loses 1 point. As soon as a post has reached a score of 0 points, it is considered inactive and is not taken into account for further analysis, even if at a later point in time, it receives additional comments. Now, the goal of the operator is to continuously compute the top-k posts with the highest score count, while the scores of the active posts dynamically change by the arrival of posts, comments and the passing of time.

The challenges in this query are splitting the event streams, handling time-outs and computing the top-k posts. In the following, we describe how we solved them in GraphCEP.

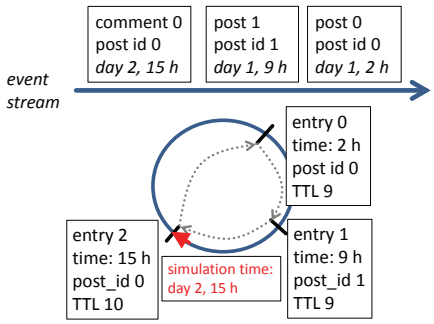


Figure 3: Query 1: Timetable.

4.1.1 GraphCEP Solution

Splitter: Stream Splitting. To split the incoming posts and comments streams, a key-based partitioning is applied. Posts are assigned to operator instances according to their post id. Each comment is assigned to the operator instance which the commented post has been assigned to. The value range of post ids, $v(O_i)$, of an operator instance O_i is determined by the residue class of the post id *id* divided by the number of operator instances n:
$id \in v(O_i) \iff id \bmod n = i$.

Operator Instance: Timeouts and Score Updates. One challenge that necessitates extensions of the existing framework is that events—comments and posts—time out, i.e., they lose score every 24 hours. A naive solution would be to generate time-out events for posts and comments and process them as an additional time-out event stream. However, this would yield a lot of additional event load. Our solution, instead, builds on an internal timetable data structure that allows for a more efficient computation of time-outs.

The timetable structure is a linked list that sorts posts and comments according to the time-of-day of their timestamp, i.e., *timestamp* `mod` *24 hours*. Each operator instance keeps such a separate timetable. Entries of the timetable contain the following information: (1) An id of the timetable entry, (2) the time-of-day of creation of the post or comment (*timestamp* `mod` *24 hours*), (3) the post id of the post (if the entry stems from a post) or of the commented post (if the entry stems from a comment), and (4) a time-to-live (TTL) counter which is decremented every 24 hours until it is 0 and the entry is removed.

As events are processed, the list is manipulated and traversed by the operator instance. To this end, the operator instance keeps a timestamp, called the *simulation time*, which corresponds to the timestamp of the last processed event. The simulation time is updated each time an event is processed. Between the old simulation time and the updated simulation time, a certain time span has passed. The operator instance uses the timetable in order to determine whether in that time span, any time-outs have happened. This is done by traversing the timetable from the time-of-day of the old simulation time until the time-of-day of the new simulation time; each 24 hours, the timetable is completely traversed once. Each entry that is visited has experienced a time-out and the score of the post that is corresponding to the entry is decremented by 1. Further, the TTL of the entry is also decremented by 1. If the new TTL is 0, the entry is removed. Finally, a new timetable entry corresponding to the newly arrived event is added to the timeable, with an initial TTL of 10. All post score updates are sent to the merger, which continuously updates the top-k posts.

Fig. 3 depicts an example. In the beginning, the timetable is empty. A post with post id 0 arrives; the simulation time is set to day 1, 2h, an entry with TTL 10 is added to the timetable, and the operator instance emits an update of post 0 having new post score 10 to the merger. Then, a post with post id 1 arrives; the simulation time jumps from day 1, 2h, to day 1, 9h, an entry with TTL 10 is added to the timetable, and an update of post 1 having new post score 10 is emitted to the merger. Finally, a comment to post 0 arrives more than 1 day later. The simulation time jumps from day 1, 9h, to day 2, 15h. In that time span, the timetable entries with id 0 and 1 are both visited: Their TTL is decremented to 9, and the post scores of post 0 and post 1 are both decremented by 1. Further, a new entry is added to the timetable with TTL 10 and the score of post 0 is incremented by 10. Finally, the updated post scores are emitted to the merger: post 0 has a new post score of 19 (decremented by 1 and incremented by 10), and post 1 has a new post score of 9 (decremented by 1).

Merger: Top-k Computation. The merger has to continuously determine the top-k scored posts, while updates of post scores are streamed by the operator instances.

Keeping all n active posts sorted by their score all the time would be relatively expensive ($\mathcal{O}(\log n)$ for processing each score update). Keeping active posts in a simple list is much cheaper ($\mathcal{O}(1)$ for processing each score update). On the other hand, (re-)computing the top-k in a sorted datastructure is easy ($\mathcal{O}(k)$), whereas it is expensive in a non-sorted datastructure ($\mathcal{O}(n \log k)$). Instead of deciding for one extreme – keeping all posts sorted or keeping none of the posts sorted – we developed a more efficient strategy based on an observation we made on the updates.

We observe that the updates of post scores are incremental; a post gains or loses a number of points over time. Depending on its score and the scores of the top-k posts, a post has a high or a low chance to be in the top-k. Based on this observation, we maintain a sorted data-structure (sorted tree) for posts with a high chance of being in the top-k, and an unsorted data-structure for posts with a low chance of being in the top-k. Posts can switch from one group to the other one when their post score is updated based on whether their score is above or below a threshold TH.

When the merger receives a score update from an operator instance, it sorts the updated posts into the sorted or the unsorted data-structure according to their new score. Then, it determines the new top-k posts: the first k posts of the sorted data-structure are the k posts with the highest score. If, however, the sorted data-structure contains less than k

entries, the missing entries have to be searched in the unsorted data-structure. Finally, the merger emits a new top-k list in case it has changed since the last update.

To avoid expensive "sorted-data-structure misses", TH is continuously adapted. The goal of adapting TH is to keep the sorted data-structure as small as possible, but never smaller than k entries. When TH is incremented, posts from the sorted data structure are transferred to the unsorted datastructure; in case TH is decremented, posts from the unsorted data structure are transferred to the sorted data structure. To limit the overhead of transferring posts between the two data-structures, the number of adaptations of TH shall be kept low. This can be achieved by changing TH in steps of higher granularity.

In the course of the case study, we found the following thresholds to work well. If the k-th highest score, $score_k$, is less than 20 points greater than TH, TH gets decremented by 5. If $score_k$ is more than 40 points greater than TH, TH gets incremented by 5. With those thresholds, the sorted data-strucure held always at least k entries.

4.2 Case Study 2: Community Detection

Finding strongly connected communities of users – whereby each user has a relation (e.g., *friendship*) to every other user in the community – currently interested in a similar topic is of major importance to social network providers and applications. In this context, our case study addresses the problem of detecting top-k comments that are liked by the largest communities of friends. More precisely, given streams of i) comments, ii) likes, i.e., users that like those comments, and iii) friendships, i.e., relations between users defining the dynamically changing friendship graph, our goal is to determine at any time t the top-k comments that are liked by the largest cliques of friends (i.e., subsets of users in the friendship graph such that the induced subgraphs are complete). Fig. 4 (top-left) shows a snapshot of the aforementioned community graph with two comments C_1 and C_2, friendship relations between users $u_1, ..., u_8$, and user likes for comments (i.e., $\{u1, ..., u_4\}$ for C_1 and $\{u_4, ..., u_8\}$ for C_2). The comment that is liked by the largest community is C_2 (with a clique size of four).

A major challenge in addressing the above mentioned query is to continuously analyse the dynamically changing communities under the consideration of multiple streams that reflect updates to the communities. For instance, each new like or friendship event can change the size of a community associated with a certain comment and therefore, can lead to a change in the top-k comments. Hence, the proposed approach, on the one hand, should be able to detect communities for multiple comments in parallel and, on the other hand, incrementally process incoming events from the streams and update the top-k list accordingly.

4.2.1 GraphCEP Solution

We used our scalable GraphCEP architecture to parallelize computation across multiple operator instances. Since a separate computation of the friendship clique is required for each comment, we parallelize this computation by assigning each comment to an operator instance. In particular, we define our implementation of the GraphCEP modules in the following.

Splitter. On receiving a *comment event*, the splitter creates a new operator instance that is exclusively responsible

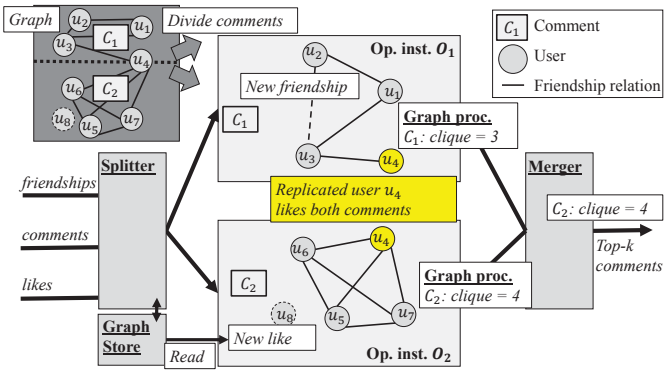


Figure 4: Community detection

for the new comment. Initially, a comment has zero likes. A *like event* is forwarded only to the operator instance that is responsible for the liked comment. A *friendship event* is forwarded to all operator instances as the new link between two users can be the missing piece for detecting a clique of a larger size for a certain comment.

Operator instance. Each operator instance O is responsible for calculating the largest clique of a certain comment C. It receives and processes like and friendship events from the splitter. Although O receives all friendship events, it discards the friendships where not both users have liked C (because the largest clique of C will not be changed by these friendships). The remaining friendship events and each like event trigger recomputation of the largest clique of C. This computation is performed using the incremental clique algorithm described below (cf. Sec. 4.2.2). If the clique has changed as a result of the incoming event(s), an event with the new clique size is sent to the merger.

Note that the problem of finding the largest clique of a graph (i.e., *maximum clique problem*) is NP-complete. Moreover, the number of likes can be extremely large (e.g. popular comments can be liked by millions of users) making it computationally infeasible for a single operator instance to detect the largest clique of friends in a timely manner. Unfortunately, operator instances can not be parallelized further by dividing the likes for C across multiple operator instances as the correctness of results will be affected. Therefore, we parallelize the operator instance by making use of a distributed graph processing system as computational module in the operator.

In Fig. 4, we give an example. Operator instances O_1 and O_2 are responsible for comments C_1 and C_2 respectively. We show the operator state that is currently maintained by both operator instances. As user u_4 has liked both comments, both operator instances maintain vertex u_4 for their friendship graph. Now, a friendship event $\{u_2, u_3\}$ is received by both operator instances. Only O_1 maintains local vertices u_2 and u_3, because both vertices only liked C_1. O_1 initiates a new clique computation and sends the clique size 3 to the merger. In the meantime, O_2 receives a like u_8. It needs to learn about all previous friendships between u_8 and any other user that has liked C_2. For this purpose, a graph store is maintained that can be contacted by all operator instances, e.g., O_2, (read-only) and is updated by the splitter (write). The triggered graph computation on O_2 leads to a largest clique size of 4 that is sent to the merger.

Merger. The merger receives events from all operator instances that have detected a changed clique size. It main-

tains a data structure for logarithmic sorted insertion of changed comments (w.r.t. clique size), e.g., a binary tree. If the top-k comments have changed because of these events, the merger issues an output event with the new top-k comments. In the case of equal community sizes, the lexicographical order of the comment text is used as tiebreaker.

4.2.2 Incremental Clique Computation

In this section, we present our algorithm for incremental clique computation on a graph $G = (V, E)$ with vertex set V and undirected edges E. We observed that new like and new friendship events can only increase the community and if the community (i.e., clique) is increased, the new clique contains a) the user that has liked the comment (in the case of a like event) or b) one of the users in the friendship relation (in the case of a friendship event). Therefore, we have to search for larger cliques only in the neighborhood of these users (i.e., graph vertices) reducing the search space significantly. In the following, we denote this vertex that was affected by the like or the friendship event as vertex x.

Algorithm 1 Incremental clique computation (sequential).

```
1: function MaxClique(C, S, θ)
2:     S = S \ {u ∈ S | degree(u) < θ − 1}
3:     if |C| + |S| < θ then
4:         return -1
5:     Λ = |C|
6:     for all s ∈ S do
7:         if ∀c ∈ C : {c, s} ∈ E then
8:             C' = C ∪ {s}
9:             S' = S \ {s}
10:            tmp = MaxClique(C', S', max(θ, Λ))
11:            Λ = max(tmp, Λ)
12:    return Λ
```

In Alg. 1, we give our recursive backtracking algorithm, that traverses the potentially very large search space of possible cliques. However, our early pruning strategies greatly reduces the search space in practice, leading to fast algorithm executions. The algorithm takes as input a current clique C, a candidate set S of possible clique vertices, and a minimal required clique size θ. Initially, the current clique contains only the changed vertex x, the set of candidate vertices S is the set of all neighbors of x, and the minimal required clique size θ is the size of the previous largest clique. In order to check, whether vertex x is part of a clique larger than θ, we call MaxClique($\{x\}, \{u | \{u, x\} \in E\}, \theta$).

First, the algorithm prunes the candidates S to contain only those vertices with a degree (i.e., the number of adjacent vertices) of at least $\theta - 1$ (line 2), because a clique of size at least θ can only consist of vertices with degree at least $\theta - 1$ (otherwise a vertex could not be connected to all clique vertices). Then, the size of the maximal clique that can be detected is the sum of the pruned candidate set and the current clique. If this size is smaller than the minimal required clique θ, clique computation is aborted, because the minimal required clique size can never be achieved in this recursive branch (line 4).

The next step of the algorithm is to determine the maximal clique recursively for all vertices in the candidate set, when adding the candidate vertex to the clique. Of course, if the candidate vertex does not have a friendship relation to all vertices in the clique, this vertex is skipped (line 7). The algorithm keeps track of the maximal clique found so far (line 11) to enable early pruning of subsequent recursion calls (lines 4 and 10). It returns the size of the maximal found clique Λ.

The above sequential algorithm is suitable for smaller graphs that can be processed on a single machine. However, as we have seen, the graph can become too large and the complexity too high for a single operator instance. Additional resources can be made available to an operator instance using a distributed graph processing system.

Algorithm 2 Incremental clique (parallel GAS).

```
1: function Gather(u, v)
2:     return D_v
3: function Sum(S_1, S_2)
4:     return S_1 ∪ S_2
5: function Apply(D_u, S)
6:     D'_u = D_u
7:     for all c ∈ S do
8:         if ∀v ∈ c ∃{v, u} ∈ E then
9:             c' = c ∪ {u}
10:            if x ∈ c' then
11:                D_u = D_u ∪ c'
12: function Scatter(D_u, D'_u)
13:    if D'_u ≠ D_u then
14:        return neighbors(u)
```

In the following, we give vertex-centric (GAS) functions for incremental clique computation (Alg. 2). These functions can be executed in parallel on each graph vertex u enabling parallelization of graph computation. As above, our goal is to find all cliques containing the newly arrived vertex x. Let vertex data D_u be the set of all found cliques containing both, vertex x and vertex u. The gather function collects data from all neighbors of u. It takes as input vertex u and neighboring vertex v and returns vertex v's vertex data, i.e., all the found cliques of neighboring vertex v (line 2). The sum function specifies how to combine gathered data. For our clique computation, the sum operation is just the union of all neighboring vertex data (line 4). In the apply function, vertex u tries to match (i.e., there is an edge between all members of the clique and u) all neighboring vertices cliques (line 8). If it can match all clique vertices, a new clique can be constructed containing vertex u (line 9). The new clique is added to the vertex data of vertex u (line 11), if it contains the goal vertex x. In the scatter function, we define the set of vertices that have to be scheduled. In case, the vertex data has changed, i.e., a new clique with vertex x is detected, all neighbors of v are scheduled for execution (line 14).

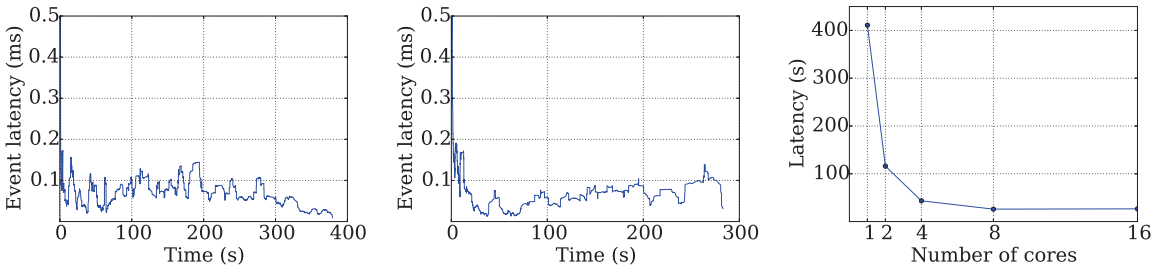
As a result, each vertex u iteratively computes all cliques, where vertices u and x participate, based on all cliques of vertex u's neighboring vertices.

5. EXPERIMENTS

In this section, we present our evaluations for the DEBS Grand Challenge 2016 using GraphCEP, and show the additional scalability of the operator by integrating modern graph processing systems.

5.1 DEBS Grand Challenge

In order to test competitiveness of GraphCEP, we participated on the DEBS 2016 Grand Challenge. Each team submitted its solution to an automated evaluation platform, a single virtual machine instance with 4 cores and 8GB of RAM. Solutions were ranked according to event latency and throughput. *Event latency* measures the delay of an event e



(a) Event latency of query 1. (b) Event latency of query 2. (c) Scalability GrapH.

Figure 5: Evaluations.

Ev./s	1	2	3	4	5
Q1	209,821	156,017	187,439	184,675	178,867
Q2	278,310	218,665	273,065	281,942	243,000

Table 1: Event throughput of query 1 (Q1) and query2 (Q2) in events per second.

that *initiated a new complex output event* e_{out}, i.e., the time between reading e from the input stream and writing e_{out} to the output stream. *Event throughput* is the total number of events that are processed per second.

We implemented our solution in JAVA with 7,200 lines of code. Because there are only 2 CPU cores for each query, a parallel approach will induce too much overhead compared to a sequential approach in terms of thread synchronization and context switches. In fact, we observed significant performance penalty, when parallelizing the queries using up to four threads per query. Therefore, we executed both queries in a single-threaded environment, enabling each query thread to run exclusively on a CPU core. This leads to extremely efficient usage of two cores without inter-thread communication, context switching and multi-threading. The other cores are used by the garbage collector and the operating system.

In Tab. 1, we give the event throughput results for different experimental runs that ranked GraphCEP among the top approaches. As input streams, we used the provided social network data with 63,409 friendship events, 435,125 post events, 742,178 comment events, and 425,936 like events. As we can see, throughput is very high for both queries (more than 150,000 events per second for query 1 and 218,000 events per second for query 2). For comparison, the average number of tweets per second issued by Twitter users is about 6,000[1]. Hence, our highly optimized solution is capable of detecting complex social network patterns with a throughput that is 36× the reported Twitter workload.

In Fig. 5a and 5b, we show how event latency evolves over time when executing queries 1 and 2. We used a large data set with 1,241,382 friendship events, 8,585,497 posts, 24,485,315 comments, and 21,594,379 likes (in total more than 55 million events). We plotted the running average over 1000 output events (y-axis) during the total runtime of the query (x-axis). As can be seen, the event latency for query 1 is not increasing over time. On the other hand, event latency for query 2 increases slightly, because of the constantly growing friendship graph. However, timeliness for both queries can be maintained, i.e., event buffers and operator instance state sizes do not fill up overwhelmingly, even for very large-scale input data. In fact, queries 1 and 2 have an average event latency of 0.046 and 0.071 milliseconds respectively. Hence, GraphCEP detects complex social network patterns in real-time.

5.2 Scalability of Graph Processing

The main idea in this paper is to increase scalability of event processing in the operator instances by utilizing modern graph processing systems. One scenario similar to above DEBS Grand Challenge 2016 is finding cliques of users who liked the same content (e.g., millions of Youtube users liking a video). In Fig. 5c, we tested scalability of distributed clique computation by executing the GAS algorithm given in Alg. 2 on the GrapH system (see Sec. 6). We performed evaluations on a shared memory machine with 32 cores (2.3 GHz AMD) and 280GB of RAM. The clique algorithm was executed on the Wikipedia who-votes-whom network[2] with 103,689 edges and 7,115 vertices. Our findings are that increasing the number of cores (i.e., subgraphs) decreases graph processing latency significantly. In fact, we observe that doubling the number of cores decreases latency by more than half. As Clique computation has super-linear time complexity, decreasing the problem size by half, decreases latency by more than half (note that this is only true, if each machine can compute Clique relatively independently on its local subgraph). This shows that utilizing modern graph processing technology to further parallelize operator instances can improve event latency significantly for graph analytics.

6. RELATED WORK

Complex Event Processing (CEP) has a vivid history in research and industry. Starting from centralized CEP systems [3], the research focus has recently shifted toward distributed CEP middleware [20, 12, 13] to reduce network bandwidth by pushing the operators closer to the source [18, 19]. As the workloads for single operators are increasing due to the proliferation of data sources, single operators became a bottleneck and concepts for operator parallelization were developed. Besides intra-operator parallelization [1], which runs different detection steps in an operator instance in parallel, data parallelization [17, 16, 10] has been proven as the most powerful parallelization approach.

In this regard, many systems have been proposed, which split the incoming event streams of an operator in different ways [17, 16, 10]. All of those systems stop the paralleliza-

[1]http://www.internetlivestats.com/twitter-statistics/

[2]http://snap.stanford.edu/data/index.html

tion at the level of operator instances which sequentially process the partitions assigned to them by the splitter component. The processing states of single partitions are not considered to be very large or even a potential bottleneck. This is where GraphCEP provides an important extension to data-parallel CEP, as it allows for massive parallelization of single operator instances when they work on graph-structured data.

In the last years, many systems for distributed graph processing have emerged that can be used as graph processing systems for GraphCEP [14, 8, 15]. A major challenge of these graph systems is to find a suitable graph partitioning strategy. For instance, GrapH [15] performed adaptive workload-aware vertex-cut graph partitioning. Recently, some systems for dynamic graph processing have been proposed [4, 6, 7, 21]. For instance, Kineograph [4] adapts the graph according to incoming tweets and performs graph analysis on graph snapshots. Fard et al. [7] address reachability and subgraph matching queries over a time-evolving graph, i.e., a series of graph snapshots. Wickramaarachchi et al. [21] enable complex graph queries over a time-evolving graph using a publish-subscribe approach (e.g., [2]) to push notifications of pattern detections to interested users. Unfortunately, all of these approaches assume a single graph and restrict the incoming data streams to contain *only* graph-structured data. In contrast, GraphCEP empowers users to utilize the great variety of complex event detection capabilities *combined* with the expressiveness and efficiency of modern graph processing systems using the GAS programming model.

7. CONCLUSION

The goal of this paper is to combine the expressiveness and timeliness of CEP with the efficiency and scalability of distributed graph processing systems. We propose GraphCEP, a CEP system with two-level scaling capabilities. First, we scale out event processing using state-of-the-art splitting techniques for independent processing on the operator instances. Second, we scale out single operator instances using recent graph processing systems. We show that GraphCEP can efficiently handle social network analytics such as activity and community detection that can not be expressed by either CEP or graph systems alone.

8. REFERENCES

[1] C. Balkesen, N. Dindar, M. Wetter, and N. Tatbul. Rip: Run-based intra-query parallelism for scalable complex event processing. In *Proc. of ACM DEBS*, 2013.

[2] S. Bhowmik, M. A. Tariq, L. Hegazy, and K. Rothermel. Hybrid content-based routing using network and application layer filtering. In *Proc. of IEEE ICDCS*, 2016.

[3] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. Telegraphcq: Continuous dataflow processing. In *Proc. of ACM SIGMOD Conf.*, 2003.

[4] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen. Kineograph: taking the pulse of a fast-changing and connected world. In *Proc. of ACM EuroSys*, 2012.

[5] G. Cugola and A. Margara. Tesla: a formally defined event specification language. In *Proc. of ACM DEBS*, 2010.

[6] D. Ediger, R. McColl, J. Riedy, and D. A. Bader. Stinger: High performance data structure for streaming graphs. In *IEEE Conf. on High Performance Extreme Computing*, 2012.

[7] A. Fard, A. Abdolrashidi, L. Ramaswamy, and J. A. Miller. Towards efficient query processing on massive time-evolving graphs. In *Int. Conf. on Collaborative Computing*. IEEE, 2012.

[8] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, 2012.

[9] V. Gulisano, Z. Jerzak, S. Voulgaris, and H. Ziekow. The debs 2016 grand challenge. In *Proc. of ACM DEBS*, 2016.

[10] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm. A catalog of stream processing optimizations. *ACM Comput. Surv.*, 46, 2014.

[11] G. G. Koch, B. Koldehofe, and K. Rothermel. Cordies: Expressive event correlation in distributed systems. In *Proc. of ACM DEBS*, 2010.

[12] B. Koldehofe, R. Mayer, U. Ramachandran, K. Rothermel, and M. Völz. Rollback-recovery without checkpoints in distributed event processing systems. In *Proc. of ACM DEBS*, 2013.

[13] B. Koldehofe, B. Ottenwälder, K. Rothermel, and U. Ramachandran. Moving range queries in distributed complex event processing. In *Proc. of ACM DEBS*, 2012.

[14] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *Proc. of ACM SIGMOD Conf.*, 2010.

[15] C. Mayer, M. A. Tariq, C. Li, and K. Rothermel. Graph: Heterogeneity-aware graph computation with adaptive partitioning. In *Proc. of IEEE ICDCS*, 2016.

[16] R. Mayer, B. Koldehofe, and K. Rothermel. Meeting predictable buffer limits in the parallel execution of event processing operators. In *IEEE Int. Conf. on Big Data*, 2014.

[17] R. Mayer, B. Koldehofe, and K. Rothermel. Predictable low-latency event detection with parallel complex event processing. *IEEE Internet of Things Journal*, 2, 2015.

[18] B. Ottenwälder, B. Koldehofe, K. Rothermel, and U. Ramachandran. Migcep: Operator migration for mobility driven distributed complex event processing. In *Proc. of ACM DEBS*, 2013.

[19] B. Ottenwälder, R. Mayer, and B. Koldehofe. Distributed complex event processing for mobile large-scale video applications. In *Proc. of the Posters & Demos Session, ACM Middleware conf.*, 2014.

[20] N. P. Schultz-Møller, M. Migliavacca, and P. Pietzuch. Distributed complex event processing with query rewriting. In *Proc. of ACM DEBS*, 2009.

[21] C. Wickramaarachchi, M. Frincu, and V. Prasanna. Enabling real-time pro-active analytics on streaming graphs. *algorithms*, 15:18.