Chronograph—A Distributed Processing Platform for Online and Batch Computations on Event-sourced Graphs

Benjamin Erb
benjamin.erb@uni-ulm.de
Institute of Distributed Systems
Ulm University, Germany

Dominik Meißner
dominik.meissner@uni-ulm.de
Institute of Distributed Systems
Ulm University, Germany

Jakob Pietron
jakob.pietron@uni-ulm.de
Institute of Distributed Systems
Ulm University, Germany

Frank Kargl
frank.kargl@uni-ulm.de
Institute of Distributed Systems
Ulm University, Germany

ABSTRACT

Several data-intensive applications take streams of events as a continuous input and internally map events onto a dynamic, graph-based data model which is then used for processing. The differences between event processing, graph computing, as well as batch processing and near-realtime processing yield a number of specific requirements for computing platforms that try to unify theses approaches. By combining an altered actor model, an event-sourced persistence layer, and a vertex-based, asynchronous programming model, we propose a distributed computing platform that supports event-driven, graph-based applications in a single platform. Our Chronograph platform concept enables online and offline computations on event-driven, history-aware graphs and supports different processing models on the evolving graph.

CCS CONCEPTS

•Applied computing →Event-driven architectures; •Computing methodologies →*Distributed computing methodologies;* **•Computer systems organization →***Distributed architectures;*

KEYWORDS

event processing, graph computing, event sourcing

ACM Reference format:
Benjamin Erb, Dominik Meißner, Jakob Pietron, and Frank Kargl. 2017. Chronograph—A Distributed Processing Platform for Online and Batch Computations on Event-sourced Graphs. In *Proceedings of DEBS '17, Barcelona, Spain, June 19-23, 2017,* 10 pages.
DOI: http://dx.doi.org/10.1145/3093742.3093913

1 INTRODUCTION

The rise of data-intensive applications has raised the bar for platforms that provide execution environments for such data-heavy applications. According to Kleppmann [16], these applications are primarily challenged by (i) the amount of data, (ii) the complexity of data, and (iii) the speed at which data is changing.

The first challenge — *the amount of data* — is widely addressed by the use of distributed architectures for scalability and reliability. *Complexity of data* is handled by sophisticated data models that are able to capture the inherent entanglements of the application domains. This includes graph-based systems for maintaining and processing complex and highly interconnected data sets. The requirement of *speed at which data is changing* has a long history in systems for event processing. However, combined with the previous two requirements, a new aspect has recently emerged: fast data. In addition to the exact outcomes of long-running, lengthy computations, the availability of approximate results within shorter periods of time has become essential for several applications.

The data-intensive applications we would like to address in this paper share a number of common properties. They consume a constant stream of incoming events from the outside world. This input continuously modifies the internal application state, modeled as a mutable, highly dynamic graph. Changes either modify the topology of the graph, or they alter the state of an edge or vertex. The graph then serves as a basis for computations on the topology. Results may in turn trigger interactions with the outside world.

For instance, online social networks maintain internal graph representations for friendship relations, contents, and interactions of its users [22]. User actions yield events that eventually modify the graph. Computations on social network graphs include traditional graph algorithms such as the identification of influential users, but also more short-term, ephemeral computations like trend analyses for contents or users.

Web crawlers construct a graph by downloading and analyzing web sites, extracting and following retrieved links, and updating the topology [4]. The resulting web graph always represents a potentially stale and outdated graph due to the fact that the crawler cannot instantly download all sites. Common computations on web graphs include the ranking of web pages or the identification of cliques (e.g., link farms). A common approach is the creation of a graph data set by crawlers, which is then used as the input for a decoupled graph computing engine. However, in an online approach, computations are executed on intermediate data and concurrently to the scraping process.

DEBS '17, Barcelona, Spain

978-1-4503-5065-5/17/06…$15.00
DOI: http://dx.doi.org/10.1145/3093742.3093913

So far, these applications require custom architectures with a high degree of complexity exposed to their developers. Different components such as event processing and graph processing systems, as well as graph databases and event logs, had to be integrated, yielding highly heterogeneous and partially incompatible programming models.

The CHRONOGRAPH platform introduces a unified platform that hides that accidental complexity and provides their users with a single system for data-intensive applications based on event-driven, dynamic graphs. In doing so, the CHRONOGRAPH platform provides solutions for the following challenges: *(i)* A stateful graph model that allows highly concurrent, local modifications of the topology, while also allowing for global, consistent views of the whole graph. *(ii)* A computational model that enables asynchronous online computations on the dynamic graph as well as decoupled batch computations thereof. *(iii)* A unified programming model that takes into account the processing of incoming event streams for the graph and vertex-centric graph computing capabilities. *(iv)* A graph persistence model that tracks the evolution of the graph over time, allowing for retrospective reconstructions and the resolution of causal dependencies.

The remainder of this paper is divided into the following sections. In section 2, we take a look at the differences between event processing and graph computing as well as online and batch computations. Next, we introduce the CHRONOGRAPH approach in section 3. The different programming and processing models enabled by this model are highlighted in section 4. The actual platform is described in section 5 and evaluated in section 6. In section 7, we compare CHRONOGRAPH to existing platforms and illustrate how our platform differs. A discussion of our results in section 8 precedes an outlook and concluding remarks in section 9.

2 BACKGROUND

CHRONOGRAPH combines concepts and mechanisms from both event processing and graph computing platforms. Furthermore, it represents a data-processing architecture that unifies online and batch computations.

2.1 Event Processing & Graph Computing

Computations on streams of events and time series data are the traditional domain of event processing systems [6]. Complex event processing engines provide an explicit topology of processing nodes. Sources inject new events into the processing network. Processing nodes then apply functions to incoming events (e.g., modifying, filtering, or aggregation operations) and may emit events to the subsequent nodes. Stateful nodes maintain a mutable state which is derived from previously consumed events. This approach is suited well for reasonably complex states such as counters or sets per node. However, in order to handle event-driven graphs, event processing systems have to store entire graph topologies as part of their processing nodes' internal states. This approach has limitations in regard to efficiency, scalability, and maintainability. Furthermore, event processing engines lack efficient support for graph algorithms and graph computations.

The necessity of large-scale graph computations and the shortcomings of general purpose batch processing platforms (e.g., MapReduce [7]) for some graph-specific computations has yielded distributed graph computing systems such as Pregel [18] and PowerGraph [12]. These systems take a stateful graph topology as an input and execute a computation on the entire graph resp. all vertices. Outcomes range from scalar results to entire output topologies. Computation tasks include the ranking of vertices based on topological metrics, the detection of structural patterns, clustering, or routing. Pregel is based on active, stateful vertices and immutable message-passing between vertices. The execution is globally stepped and follows a bulk-synchronous-parallel execution model. PowerGraph, instead, represents an asynchronous approach and applies the gather/apply/scatter pattern. States are stored on edges and can be asynchronously accessed by adjacent vertices. Both systems are based on a vertex-level programming model. Alternative models for distributed graph computing include sub-graph-centric approaches or partitioned tables [15]. These approaches impose less access locality restrictions compared to vertex-centric models, but come with the cost of increased coordination. The notion of dynamic graphs that may change over time has also been partially addressed by recent graph platforms (e.g., [14, 21, 26]). In section 7, we introduce more graph platforms when we differentiate CHRONOGRAPH from existing work.

The CHRONOGRAPH platform follows traditional graph computing systems by using a vertex-centric model, but at the same time it incorporates the perspective of an asynchronous processing network for handling streams of events and updates.

2.2 Online and Batch Computations

The necessity to simultaneously provide exact results and faster, approximate, preliminary results has yielded architectures that try to combine both computational models [10, 19]. The basic idea of the *Lambda architecture* is the separation of two distinct processing layers, both of which receive the same data as input. The batch layer occasionally applies batch processing on the entire data set for computations. By contrast, the speed layer constantly applies stream processing for incoming data and provides faster, but less accurate results. Results of both layers are typically stored in a distributed data store. A serving layer provides querying capabilities and brings together both result types, depending on their availability. The *Kappa architecture* rejects the use of different and incompatible processing models. It is solely based on event processing and event replay mechanisms. Thus, the Kappa architecture replaces the batch processing jobs of the batch layer with multiple, parallel stream processing jobs instead.

In short, the CHRONOGRAPH platform can be seen as a hybrid architecture, but customized for graph-based applications.

2.3 Event Sourcing

The ordinary way of dealing with mutable application state updates latest states by overwriting old states and replacing them with new states. Event sourcing [11] is an architectural style for the state persistence of event-based systems which takes a different route. Event sourcing appends all events that yield state changes to an event log. This log preserves the entire history of the application

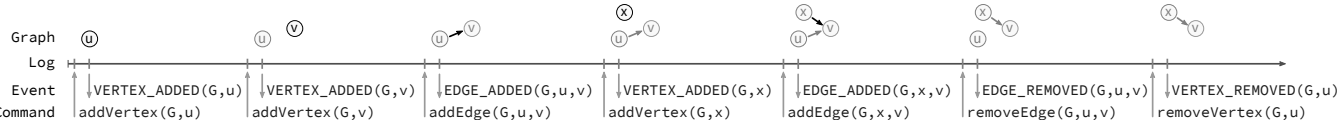


Figure 1: A central log for sourcing commands and events of graph topology changes. Previous graph states can be reconstructed by replaying event sequences. Note that CHRONOGRAPH applies a distributed variant with vertex-local event logs.

state. The latest state can be seen as a left-fold operation of all event log entries. Not only the current, but also previous states can be reconstructed by re-applying the events from the log up to a certain log entry. Note that event sourcing can either be applied to the global application state, or it can be used for the state of each application entity individually.

Distributed, event-based architectures often apply event sourcing in conjunction with Command Query Responsibility Segregation (CQRS) [27]. This architectural style separates read and write operations for domain objects. A command interface processes update operations on the state, whereas a query model allows for read-only access to the latest state. Event sourcing and CQRS are a good match, as successfully executed commands yield exactly the state-changing events that have to be appended to the event log. When also applying command sourcing, the command that has triggered the state-changing event is also logged and prepends the actual event. Command sourcing enables interesting possibilities for replaying application executions. Instead of reconstruction states, stepping through logged commands allows for an actual re-execution of command handling logic.

Event sourcing is one of the major enabling concepts of CHRONOGRAPH and is used as its main persistence mechanism.

2.4 Event-sourced Graphs

Event sourcing can be used to track the evolution — and consequently, the history — of mutable graph topologies [8]. The basic commands for topology modifications of a directed graph G include: `addVertex(G,x)`, `removeVertex(G,x)`, `addEdge(G,x,y)`, and `removeEdge(G,x,y)`.

A successfully executed command yields a corresponding event, e.g., `addVertex(G,x)`→`VERTEX_ADDED(G,x)`. Commands can also be rejected by the command processor due to unsatisfied preconditions yielding empty events. Edges can only be spawned between existing vertices, and vertices cannot be removed as long as they have adjacencies. By left-folding all events of the event log of a graph, the current topology of a graph can be constructed, as depicted in Figure 1. In terms of CQRS, a queryable graph representation can be maintained by always applying new events to the latest graph state whenever they become available.

When the vertices and edges of the graph maintain values, the modification of vertex or edge states can be tracked as well: `addVertex(G,x,s)`, `addEdge(G,x,y,s)`, `setVertexState(G,x,s)`, and `setEdgeState(G,x,y,s)`.

Vertices and edges then preserve an internal state s that can be altered. Again, the actual state of a vertex or edge at a certain point in time can be derived by applying all previous state-changing events of that entity.

2.5 Distributed Event-sourced Graphs

For CHRONOGRAPH, we decided against a fully global event log, because it hinders concurrency and always forces sequential graph operations. In an alternative approach that we use for the remainder of this paper, each vertex maintains an individual event log, and most operations are bound to the local context of the vertex. The global event log only remains necessary for tracking the creation and removal of vertices. The creation of a vertex triggers the provision of a new vertex event log. The state of a vertex is entirely bound to its event log, as each state-changing event can be appended locally. By changing the semantics of edges — an edge is always bound to its *outgoing* vertex — edges can also be stored as part of the local state of a vertex. When using individual logs, the state of any vertex can be reconstructed solely by processing its own log. For reconstructing the entire topology, the global log and all individual logs are required.

CHRONOGRAPH is based on a event-sourced graph that uses individual logs for each vertex and a shared event log for global topology events (i.e., spawning vertices and marking vertices as disabled). In addition, the graph is restricted to be directed and must not contain any self-loops.

3 THE CHRONOGRAPH APPROACH

In order to enable highly parallelized, distributed computations on the graph, we opted for an asynchronous computational model in which each vertex can individually execute behavior and communicate with adjacent vertices using message passing. At the same time, we added a versioning mechanism to the messages and event logs in such a way that we can use our event sourcing capabilities to reconstruct global and consistent graph snapshots. These "freezed" graph copies are then used for a number of decoupled processing operations, independent from the evolving live graph.

3.1 Execution Model and Concurrency

Based on our requirements regarding an asynchronous and highly concurrent graph model, we decided to superimpose the actor model onto the event-sourced graph model.

The actor model [2] is an execution model that inherently provides concurrency and distribution properties. Its primitives, lightweight processes that are called actors, can communicate with other actors using asynchronous message passing. On receiving a message, an actor can (i) send messages to other actors, (ii) spawn new actors, or (iii) modify its internal state or behavior (i.e., the logic executed upon receiving the next message). The model inhibits global state and only allows for isolated, local states of actors. For messaging, immutability is enforced. Based on these restrictions,

the actor model provides interesting properties for asynchronous systems that facilitate distribution, scalability, and responsiveness.

In CHRONOGRAPH, each actor represents a vertex. The local state of the vertex becomes the internal state of the actor. Unlike the actor model, in which messages can be addressed to any other actor, we limit message passing of a vertex to its adjacent vertices (along edges) and a dedicated entity representing the system. The actor model provides no message ordering except for FIFO ordering of messages having the same sender and receiver. This corresponds to FIFO ordering along an edge in our graph. In CHRONOGRAPH, a vertex can react to an incoming message by executing zero or more of the following four operations as part of its internal behavior:

(i) *Internal state updates*. Based on the current state, the current behavior, and the contents of the received message, the vertex can execute local computations and update local values. (ii) *Behavior modification*. The vertex can modify its behavior for the next message to be processed. (iii) *Message dispatches*. Messages can be sent via all outgoing edges. (iv) *Topology changes*. Changes of the topology include spawning new vertices or adding edges. These operations transparently yield messages to the system entity.

Once a behavior function has been executed, the issued operations are performed thereafter. In terms of event sourcing and CQRS, incoming messages correspond to commands, whereas internal state updates or dispatched messages correspond to resulting events. Hence, each incoming message (i.e., command) is followed in the event log by a (composite) event that encapsulates all operations that have been performed. Note that a message is part of an event of the sending vertex while at the same time representing a later command for the receiving vertex. Due to immutability and asynchronicity of the messages, the log entry is duplicated and appears in the log of both the sender and receiver.

3.2 Versioning of Vertices

So far, the graph follows an eventually consistent approach, in which vertex-local updates are always visible to the vertex itself, but no globally consistent view onto the topology is available on a permanent basis. Therefore, we add lightweight versioning to the individual event log entries and messages that allows for the construction of global snapshots.

Each individual log maintains a local, scalar clock value that is incremented for each incoming message. On processing the incoming message m_{in} at local clock value t of the receiving vertex, the vertex executes its current behavior function f_t. The behavior is called with the two parameters m_{in} (i.e., received message) and S_t (i.e., current state of the vertex) and yields three results: the updated state S_{t+1}, the future behavior function f_{t+1}, and a list of $0 \ldots n$ messages m_{out} to be sent to other vertices:

$$(S_{t+1}, f_{t+1}, [m_{out}, \ldots]) = f_t(m_{in}, S_t)$$

Note that m_{in} represents the command, while the tuple of S_{t+1}, f_{t+1}, and $[m_{out}, \ldots]$ is the corresponding composite event.

Each sent message contains the scalar clock value of its sender, piggybacked to the actual message payload. Upon processing a message, the receiving vertex reads the sender's clock value and updates its local vector of the latest clock values known. Hence, each vertex holds a vector of vertices it has received messages from and their latest local clock values known to the vertex. The vector is part of the local state of a vertex and not mutable by the user. Unlike traditional vector clocks, this versioning vector does not describe a system-global state. It is rather a set of vertices and interactions the current state of the vertex is causally dependent of. The local clock values and versioning vectors are the enabling mechanisms for the construction of consistent and global topology views.

3.3 Vertex Types

In CHRONOGRAPH, there are two different types of vertices. *Stateful, event-sourced vertices* populate the core of the graph, whereas *I/O vertices* are located at the boundary of the graph and connect the graph to the outside world.

Stateful event-sourced vertices form the primary primitives of an application, capturing local state, maintaining local topology information (i.e., edges), and providing application logic. In this way, the online computations, the graph topology, and all application states are collapsed into a single graph model. Hence, a vertex behavior function can provide multiple functionalities such as handling topology change messages and executing vertex-specific application logic. Vertices are implicitly typed by their initial behavior function and state, and they are identified using a unique, internal ID. Due to the fact that a vertex can only update its state and dispatch messages once it has received a message, its behavior is purely reactive. Furthermore, the behavior function of a vertex is limited to a single-threaded, sequential execution of messages per vertex. The function only allows operations free of side-effects and is limited to the local vertex state. This implies that a vertex can neither use network communications, nor can it access local files. Communication is restricted to message passing, and event logging is provided by the runtime environment, transparent to the vertices. For each vertex, state is strictly limited to an internal state, which is also managed transparently. These restrictions combined with the asynchronicity of messaging may seem like a rather weak vertex model at first. However, a lot of distributed graph algorithms can be directly implemented with these assumptions.

With *I/O vertices*, we introduce a special kind of vertices that can interact with the outside world by consuming or producing external events. I/O vertices are part of the topology, but are placed at the graph boundary. While the inner topology of regular vertices represents the side-effect free part of the system in regard to functional programming terminology, the I/O vertices connect that inner part to an outer part prone to side effects. Compared to traditional event processing pipelines, I/O vertices correspond to sources and sinks of streams. Therefore, I/O vertices are not subject to the programming model limitations of regular vertices. At the same time, the internal states of I/O vertices are not captured by event sourcing mechanisms. Event logs of I/O vertices only append incoming messages from and outgoing messages to regular vertices. Each I/O vertex is linked to one ore more connectors (i.e., external processes) outside of the system, which in turn can carry out network communications or utilize I/O resources. This includes operations such as writing data to local files, dispatching HTTP requests, accessing database systems, or interacting with remote services via APIs. Input vertices wait for external stimuli, receive

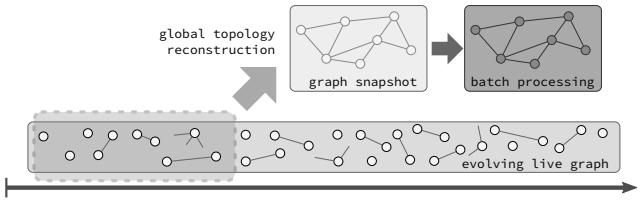


Figure 2: The live graph tracks changes to the evolving graph in local event logs of its vertices. A snapshotting mechanism uses the event logs to provide global graph snapshots, which in turn could be used for decoupled batch computations.

them, process them, and eventually assemble internal, application-specific messages to be sent into the graph. In turn, output vertices that receive messages from the regular vertices of the graph trigger external operations.

3.4 Snapshotting Mechanisms

In order to decouple global computations from the asynchronously and concurrently evolving live graph, we take advantage of the event sourcing capabilities to construct global views of the graph — either a reconstruction of a distinct, past state of the graph (*retrospective snapshot*), or a graph snapshot close the current system state (*live snapshot*). A snapshot is defined as a list of all stateful vertices and their corresponding local clock values that collectively provide a global, consistent state, as shown in Figure 2. In terms of event-sourced graphs, *causally consistent* refers to the property of a snapshot that does not contain a vertex that has received a message which its sender has not yet sent. I/O vertices are not taken into account as their reconstruction semantics would be elusive.

The computation of *retrospective snapshots* represents an expensive operation, but can be decoupled from the running application. Our previous work [9] introduced retrospective snapshotting mechanisms and optimizations for distributed event-sourced systems.

Live snapshots capture a state of the graph with minimal staleness. As the CHRONOGRAPH execution model is highly concurrent and asynchronous, the staleness of a live snapshot competes with the liveliness of the evolving graph. While pausing the entire system would allow for the retrieval of the global graph state, it interferes with our liveliness properties. We opted for a less intrusive approach in which we maintain and asynchronously update a latent snapshot (see section 5). Alternatively, an adapted version of a snapshotting algorithm by Lai et al. [17] could be applied for each snapshot epoch. The latest result of live snapshotting is the default input used when dispatching batch operations, as it represents the least stale global version of the graph. Periodically retrieved live snapshots can also be used to run iterative batch operations.

3.5 CHRONOGRAPH Applications

A CHRONOGRAPH application is defined by a set of vertex behaviors, I/O vertices and connectors, and a bootstrap vertex which spawns the initial topology. Bootstrapping typically includes the creation of several vertices and edges, as well as spawning I/O vertices which in turn establish a connection to their external connectors. Once the initial topology is set up and all initial messages are processed, the graph eventually reaches a steady state. The graph can only evolve once external stimuli arrive, i.e., when I/O vertices receive external streams of events which are then forwarded as messages into the inner graph.

A minimal CHRONOGRAPH application only relies on the online programming model with asynchronously communicating, active vertices. In order to support more batch-oriented processing of the graph, applications can be complemented by a different computations following the models in section 4. These computations can be submitted ad-hoc at runtime, but can also be provided as part of the initial application.

4 PROGRAMMING & PROCESSING MODELS

In this section, we provide an overview of programming and processing models that are supported by our event-sourced graph computing approach (also see Table 1). For each model, developers implement user-defined functions using the provided APIs.

4.1 Model for Live Graph Computations

The main programming model for CHRONOGRAPH is based on a vertex-centric model with concurrently active, stateful vertices and asynchronous message passing, as introduced earlier. The model is a natural fit for distributed asynchronous graph algorithms and algorithms that apply patterns such as scatter-gather. Upon receiving external stimuli from input vertices, the graph vertices can exchange messages, compute and update their local states, and adapt their surrounding topologies. Eventually, the graph may also yield external messages to the outside world using its output vertices.

The programming API for regular vertices is similar to that of an actor-based programming language, but with additional vertex-specific functions and limitations:

```
// event-sourced vertex behavior function
function vertexBehavior(message: incoming, state: currentState)
  sendMessage(vertex: neighbour, message: content);
  spawnVertex(function: behavior);
  spawnEdge(vertex: target);
  removeEdge(vertex: target);
  listOutgoingEdges();
  shutdownVertex();
  return newState;
```

The API for I/O vertices is less restricted, as I/O vertices are required to handle different types of communications with external processes. For input vertices, the `sendMessage()` is provided for disseminating data into the graph. Output vertices implement the callback function `onMessageReceived()` for handling incoming messages from other vertices of the graph.

4.2 Models for Offline Computations

Offline computations are local or global computations that are decoupled from the live graph execution. Instead, offline computations take a snapshotted version of the graph as input and use a *copy* of its state. We currently support different types of MapReduce-based batch computations and Pregel-like iterative graph processing operations. Furthermore, replaying of vertex event log entries provides an event-based processing style which is scoped to the time series of single vertices.

The *vertex-based MapReduce* computation is very similar to the original MapReduce interface [7] by submitting the state of each

Table 1: An overview of different programming and processing models supported in the CHRONOGRAPH concept.

Model	Initial Data Set	Locality[a]	Time Model[b]	Description	Ref.
CHRONOGRAPH	live graph	vertex-local	continuous	The default CHRONOGRAPH model for online computations on the live graph uses vertex-local computations on receiving asynchronous messages from adjacent vertices.	—
MapReduce (vertex-based)	graph snapshot	vertex-local	snapshotted	This model takes a set of all vertex states of a graph snapshot as an input and executes a MapReduce function on these states.	[5, 13]
MapReduce (edge-based)	graph snapshot	vertex-local	snapshotted	The edge-based variant provides for each edge the states of two adjunct vertices of that edge. Hence, the map receives an ordered pair of vertex states.	[5, 13]
Pregel	graph snapshot	vertex-local	snapshotted	The Pregel model takes a stateful graph snapshot as an input and applies a graph computation based on vertex-local logic and synchronous message passing.	[18]
Event Folding	log sequence	single-vertex	time series	An aggregator function left-folds a chronological sequence of state-changing events that have been logged for a given vertex.	[11]
Command Folding	log sequence	single-vertex	time series	A chronological sequence of commands (i.e., incoming messages) is left-folded for a given vertex.	[11]
MapReduce (temporal)	graph snapshot	vertex-local	snapshotted/ iterative	The temporal MapReduce variant provides two states of a vertex as an input to the map function, each coming from different snapshots.	—
Pause/Shift/Resume	graph snapshot	vertex-local	iterative	An offline execution is executed on a current snapshot. Once a more recent snapshot is available, the execution is paused, the intermediate changes are applied, and the computation is resumed.	[14]

[a] *Locality:* vertex-centric (vertex-local state and/or behavior, global execution); single-vertex (vertex-local state and behavior, execution limited to single vertex).
[b] *Time Model:* continuous (online graph); snapshotted (retrospective/asynchronous live snapshot); time series (event log sequences); iterative (multiple live snapshots).

vertex of the snapshot to the map function [5]: `mapVertex(state)`. The map function may `emit()` zero or more intermediate key-value pairs, which are then sorted by key. For each unique key, the `reduce(key, values)` function receives a list of values emitted for that key. This model is especially useful for non-iterative computations on the states of the vertices, such as global aggregations and reductions. The MapReduce interface can also be used for generating ephemeral graph views (e.g., filtering sub-graphs).

The *edge-based MapReduce* computation executes the map function on each edge instead [13]. As the edges of CHRONOGRAPH are not stateful, states of the outgoing and incoming vertices of the edge are taken as inputs for the map function: `mapEdge(vertexOut, vertexIn)`. Note that this mechanism is more demanding in terms of resources, as the state of a single vertex may be used as in input multiple times (i.e., for each edge of a vertex). This interface is primarily useful for edge-centric computations when access to the states of both vertices is required.

The *Pregel* model [18] shares similarities to our online computation model, but relies on synchronized supersteps for messaging. Global and explicit message rounds allow a straightforward implementation of many distributed graph algorithms: The Pregel model represents a good batch processing counterpart to our online model, which focuses on liveness and mutability of the graph instead.

The command and event folding operations are based on the vertex event logs and the replay of its entries (i.e., commands and reconstructed states). For *event folding*, a folding function is applied to the sequence of events. For the event e_t, the function receives the states at e_{t-1} and e_t and an aggregate state that is then passed to the next invocation: `fold(oldState, newState, aggregate)`. Event folding allows for time series computations over vertex state over time. By passing the previous state into the aggregating function, the two different states can be compared as part of the fold operation.

When executing *command folding*, the aggregating function is applied to the sequence of messages received so far: `fold(command, aggregate)`. Command folding can be used to simulate an alternative behavior function. When used as a behavior function (without sending messages or modifying the topology), the fold function maintains its vertex state as the aggregation state.

4.3 Hybrid Models

Hybrid models execute batch-oriented computations, but also take into account the evolution of the graph.

The *temporal MapReduce* takes two snapshots as an input. For each vertex, the reconstructed states are passed to a map function: `mapTemporal(state1, state2)`. If a vertex has not yet existed or has been shutdown in a snapshot, an empty state is used. This

function allows to match different states of the same vertex and emit key-value pairs based on the comparison.

The *Pause/Shift/Resume* approach of GraphTau [14] is an extension to a bulk-synchronous Pregel-like API for handling incremental processing on evolving graphs. The computation starts with an initial graph snapshot. After each superstep, the runtime checks whether a new snapshot is available. If so, the execution is *paused*. Next, the computation is *shifted* to the new snapshot. An algorithm-specific function updates the topology and reckons up intermediate results. Finally, the computation is *resumed* and the next superstep starts. We argue that this concept is a good match for event-sourced graphs. Recurrently triggered live snapshots can provide a stream of periodical graph snapshots. The `shift()` function of a computation can be augmented by the event log of a vertex when taking into account the sequence of events between both snapshots.

5 CHRONOGRAPH PLATFORM

We implemented a research prototype of the CHRONOGRAPH platform to demonstrate its feasibility and to evaluate the concept.

5.1 Architecture

The CHRONOGRAPH platform is designed as a distributed platform that shards the graph to a fixed number of worker instances. Each worker is responsible for message handling, execution scheduling, state handling, and event logging of its vertices. Hashing is used to assign vertices to worker instances based on their internals IDs.

Conceptually, the platform is split into several layers: (i) a distributed infrastructure with a reliable messaging system; (ii) an actor-based runtime platform with event sourcing capabilities; and (iii) the actual graph computation layer as a library layer on top of the actor platform. Due to the lack of event-sourced actor systems we could have built our system on, we designed and implemented a lightweight actor platform by ourselves. For communication, workers use a distributed messaging system which provides message exchange according to the messaging semantics (i.e., eventually guaranteed delivery, FIFO ordering between actor pairs). A secondary messaging channel between all workers is used for management communication and coordination. The workers follow a shared nothing-design, meaning there is no shared state between workers. Vertex-internal functions such as spawning vertices are transparently mapped to internal messages sent to a "system" entity. These messages are then received and handled by the worker. Hence, topology modifications are tracked by the causing vertex and the worker-local log. Each worker provides a main component that handles the live actor topology and additional modules for decoupled computations. We decided to design the main component based on a single-threaded event-loop. On each loop, the worker pulls new messages from the messaging system, deploys them to the inbox queues of the corresponding vertices, and handles the queued messages by executing the local actor behavior. The execution may in turn trigger the dispatch of new messages to the messaging system. Workers also store the latest states of their vertices in-memory, and manage the persistent event logs for their shard. For lightweight snapshotting, we added a hierarchical versioning mechanism that also tracks the workers. Each worker maintains a worker-local, logical clock value which gets

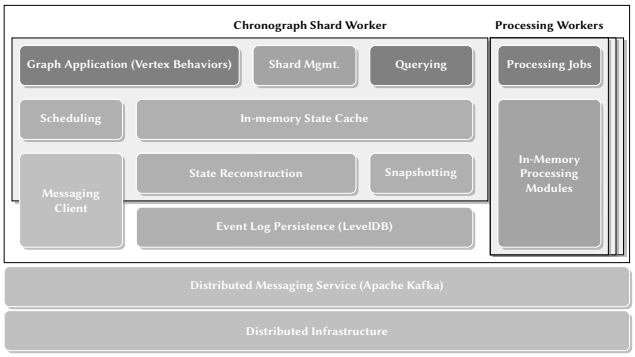


Figure 3: Internal architecture of the Node.js-based CHRONOGRAPH worker instances responsible for handling shards of the graph. A dedicated worker processes the shard-local live computations and additional processing workers provide the other programming and processing models.

incremented whenever one of its vertices increments its clock. This worker-local clock value t_y is piggybacked to the vertex clock value t_{x_i} of vertex i in the event log and for messaging: (t_{x_i}, t_y). Due to the single-threaded worker design, the worker clock value is sufficient to define the corresponding states of all vertices of that worker. By also piggybacking the latest worker-local timestamps to messages to remote workers, each worker can keep track of the other workers and their local timestamps. A vector of all workers' timestamps provides a concise snapshot identification for the entire system. Given an arbitrary worker y with target clock value z, all of its vertices $x_i \in Shard_y$ are reconstructed by selecting the latest log entries (t_{x_i}, t_y) that satisfy $t_y \leq z$. For retrospective snapshots with a target vertex and a vertex-local target clock value, the corresponding clock of its worker has to be retrieved first as part of the meta data of an event. Next, the worker reconstructs its vector of other worker clocks at that time. Finally, the worker issues a reconstruction task to the other workers with their corresponding clock value.

In order to speed up reconstruction operations, the vertex states in the event logs can be checkpointed by periodically inserting a full vertex state into the event log.

5.2 Implementation

Figure 3 illustrates our actual worker architecture. We decided to implement the platform workers in JavaScript (i.e., ECMAScript 6) using Node.js[1] as the runtime environment. In total, the current platform prototype consists of 2,305 lines of code plus several JSON-based configuration files. We also utilized several npm packages[2]. CHRONOGRAPH applications run by the platform consist of JavaScript functions for describing vertex behavior and the other computations. User code is always executed in a sandboxed environment. Each worker is executed as a single-threaded process and maintains the evolution of its graph shard at runtime using the CHRONOGRAPH model for live graph computations. The I/O vertices are implemented as programmable endpoints that can communicate with arbitrary external processes using TCP connections.

[1] https://nodejs.org/
[2] https://www.npmjs.com/

At startup, each worker responsible for a shard spawns additional child processes that provide the other processing and programming models locally for that shard. These processing worker instances communicate with their main shard worker process (e.g., for retrieving snapshots) by using the messaging abstraction of Node.js for inter-process communications. While the evolution of the individual vertices is persisted in the event log for the live graph by the shard worker, its processing workers only work on an in-memory copy of their shard. Processing workers use a separate instance of the distributed messaging service to exchange results and coordinate computation steps. We maintain vertex states as JSON data and apply the JSON Patch format (RFC 6902) to capture the change operations of state changes, provided by the npm module fast-json-patch. JSON is also used as the native format of the messages between vertices. For messaging, we deployed Apache Kafka[3] and kafka-node (npm) as connector. A dedicated topic is used for each worker, which guarantees FIFO semantics and separates communication channels between workers. The topic of a worker is used whenever a message has to be delivered to a vertex of that worker's shard. No naming or location service is required because of the hash-based allocation mechanism of vertices. Event log persistence is provided by LevelDB[4] and the npm modules levelup and leveldown. Each worker uses a single database for all event logs of its vertices. The key of an entry encodes the vertex ID, the vertex clock value and the entry type (i.e., command, event). Additional database files are used for persisting other data such as snapshot vectors. Workers provide a HTTP-based API for external read access to local vertex states, snapshots, and batch operation results. Furthermore, new snapshots can be triggered and new computing tasks can be submitted by using this API.

6 PLATFORM EVALUATION

For testing our prototype, we implemented example applications for the use cases introduced in section 1. We then evaluated our platform implementation by measuring several metrics.

6.1 Example Applications

For each application, we prepared an external stream of events a priori. We opted for an emulation-based streaming in order to obtain reproducible results. An external process replays the events for each evaluation run and streams the data into the platform via I/O vertices.

In the *Online Social Networks* example, we went back to the data sets provided for the 2016 DEBS Grand Challenge[5]. The `friendships.dat` stream contains a sequence of events representing friendship changes in a social network. While replaying this stream, we constantly updated the social graph topology. The topology consisted of vertices representing the users, apart from the input vertices and a set of vertices that were responsible for spawning and updating the topology. We transformed the undirected friendships into bidirectional edges for our directed graph. The raw graph contained 42,934 vertices and 1,241,381 (undirected) edges. We computed the connected components using a bulk-synchronous algorithm on graph snapshots. We used iterative snapshots on the evolving graph to compute the average number of friends per user with a vertex-based MapReduce function.

Table 2: Time required to snapshot/reconstruct a graph of from the web graph workload (60,826 vertices on 8 workers).

	Average	SD
Graph snapshotting	3.65 ms	0.80 ms
Graph reconstruction	2701.48 ms	64.31 ms
Event restore rate	75728.49 events/s	1766.01 events/s

For the *Online Web Crawling* example, we used a smaller, prefetched web graph[6] (60,826 vertices, 143,766 edges). Instead of replaying event streams, the external process emulated request/response exchanges by providing prepared responses for each site in the web graph. Each request/response cycle was configured with a uniform latency of 500 ms. We provided an offline and online PageRank computation. The former was based on the default algorithm [3] and utilized the bulk-synchronous model inspired by Pregel. For online computations, we applied an asynchronous approximation running on the live model of CHRONOGRAPH, which was adapted from a distributed PageRank variant [25].

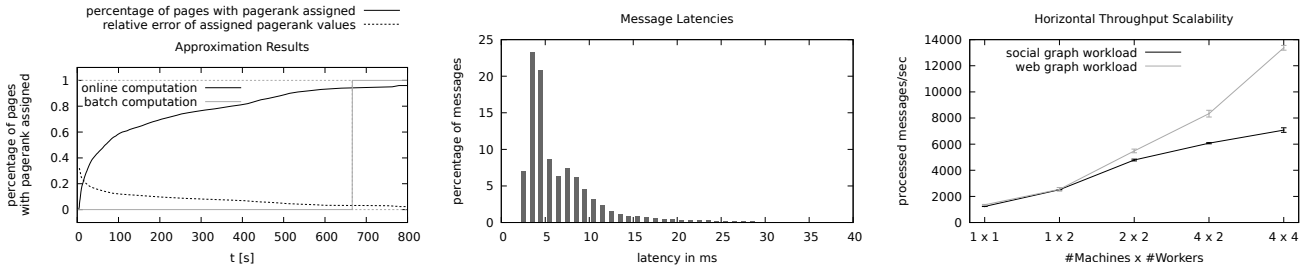
6.2 Performance Measurements

The measurements were conducted on four identical dedicated machines, each equipped with an Intel Xeon E31220 CPU (quad-core; 3.10 GHz), 16 GB of memory, and 1 GigE NIC. The four machines were connected in an isolated, Gigabit Ethernet local area network. Ubuntu 14.04 LTS was used as the operating system with Linux Kernel 3.16. We installed version Kafka 0.10.2, Node.js at version 7.5.0, and used LevelDB 1.19.

We executed the example applications with different workloads and different platform configurations (i.e., worker counts and machine counts), as shown in Figure 4. In order to assess the CHRONOGRAPH concept, we focused our measurements on properties of the online graph computation. The performance of re-implementations of existing data processing mechanisms was not part of our evaluation. For vertex messaging, we measured throughput and latency. We also considered the scalability of the platform with different amounts of workers and machines. Regarding snapshots, we measured the time required to compute and select consistent snapshots, and the time necessary to reconstruct a corresponding graph copy. All measurements were repeated three times.

The web graph application maintained a growing topology and its vertices were incrementally ranked with an online algorithm. As shown in Figure 4a, we tracked the number of web pages in the graph that already had been assigned an approximate rank value. At the same time, we retrospectively computed the relative errors of that approximations. While the graph evolved, the number of ranked pages increased an the relative errors decreased due to convergence of that algorithm. Note that approximate results were available while the web graph was still being scraped. These provisional results would not have been available in traditional batch platforms, as depicted by the offline result available later.

[3] https://kafka.apache.org/
[4] http://leveldb.org/
[5] http://www.ics.uci.edu/~debs2016/call-grand-challenge.html
[6] http://snap.stanford.edu/data/



(a) Online and offline ranking computations on an evolving web graph. The online computation variant provides increasingly accurate approximative results early on.

(b) Distribution of latencies in the social graph stream workload for messages between vertices. Average latency was 5.02 ms, 95th percentile 15.16 ms, 99th percentile 29.18 ms.

(c) Global message throughput for two workloads on different setups. The web graph application benefited more from additional workers than the social graph application.

Figure 4: Different performance measurements taken with various workloads of the example applications.

Figure 4b shows typical latencies of application messages sent between vertices of the graph. As shown in Figure 4c, highly concurrent applications benefited from additional instances in particular.

Table 2 lists the time required to compute a consistent graph snapshot and reconstruct all of its vertex states for 60,826 vertices from the web graph. Decoupled batch operations on a reconstructed graph copy could have been started in less than three seconds after the snapshot had been initialized.

7 RELATED WORK

Traditional batch processing solutions for distributed graph computing take a fixed graph as input and execute a predefined computation. Pregel [18] applies a synchronous, super-stepped, and vertex-based model while PowerGraph [12] allows asynchronous operations on an edge-centric model. Unlike these systems, CHRONOGRAPH allows the graph to evolve based on external events while supporting online processing.

Popular general-purpose frameworks for distributed data processing often provide graph-specific processing libraries, such as GraphX for Apache Spark [13] or Gelly for Apache Flink [1]. Both offer a number of different graph computing models, including a Pregel-like API. However, neither GraphX nor Flink are designed as dedicated application platforms for graph-based, event-driven applications that run continuously. Also, the inherent support for temporal graph analytics and built-in data lineage (i.e., event log persistence) distinguishes CHRONOGRAPH from both libraries. GraphTau [14] is an extension to the bulk synchronous model of GraphX for evolving graph processing. We adopted the mechanism to our concept. For snapshotting, GraphTau relies on NTP-synchronized workers while we employ causally consistent snapshots based on the event logs.

GoFFish [26] is a graph analytics framework for batch processing on graphs with slow-changing topology. GoFFish augments a bulk synchronous parallel model and takes advantage of subgraph-centric computations. GoFFish targets offline bulk processing on large datasets though and is not designed for dynamic topologies and streaming data. ImmortalGraph [21] and its predecessors represent systems for storing and analyzing temporal graphs. ImmortalGraph provides a query interface, an iterative computing interface, and several optimizations for temporal graph persistence and scheduling. However, it does not support the interactive mode of CHRONOGRAPH in which the application graph can continuously interact with external systems.

Naiad [23] is a low-latency, iterative dataflow system that shares several similarities with CHRONOGRAPH: both systems feed a stream of events into an active, directed graph in order to provide iterative and incremental computations. Being a generic dataflow system, the graph of Naiad only represents the dataflow execution topology, while we explicitly use the graph topology also as the actual state of a graph application.

Eventuate [24] is a microservice architecture that shares some similarities with CHRONOGRAPH as it also applies event sourcing, CQRS, and the actor model. Eventuate specifically addresses issues of event-driven, distributed data management in decentralized, replicated systems instead of event-driven, distributed graph processing in a centralized system. The usage of event sourcing and CQRS differs in Eventuate, as it provides more specific abstractions to the developer (actors, views, writers, and processors). In CHRONOGRAPH, the main abstractions are collapsed into the vertex entities and commands are intertwined with vertex message passing.

8 DISCUSSION

We are aware that the CHRONOGRAPH concept is not a silver bullet for graph-based applications which run on external event streams. Neither the live programming model nor the replay-based folding operations do match up to dedicated event processing solutions. Specific batch processing solutions with a single model and a highly optimized implementation will always outperform our hybrid, multi-model approach. We still believe that our concept is relevant for applications when the benefits of a seamless, unified system outweigh performance penalties.

Most graphs that we target could be executed on a single machine regarding their reasonable topology scale. A single machine solution may even be beneficial in terms of performance for a batch system [20]. We still designed our CHRONOGRAPH prototype as a distributed system due to the degree of parallel vertex behavior executions. More workers increase the execution parallelism and hence also the the liveness of our reactive system.

The usage of event sourcing comes with significant storage requirements, as the progress of the application is persisted in the event logs. However, data lineage and data provenance requirements are becoming more important for several data-intensive

applications which log most of their application data anyway. The CHRONOGRAPH concept inherently satisfies the requirements by event sourcing and allows for retrospective application state reconstructions and explorations (e.g., post-mortem analyses).

The model allows for event log pruning. When a global snapshot is computed all events prior to the snapshot can be folded into a single event. This approach loses any history and causality information prior to the snapshot, but can bound event log growth.

Most temporal graph processing systems capture the graph partitions from the worker instances based on a best-effort approach or a (synchronized) wall clock value. Instead, we opted for consistent snapshots that takes into account our message-based programming model and the local clock values of the vertices. Hence, we trade some staleness of snapshots for the guaranteed consistency thereof. Based on the evolving live graph, any snapshot has to be regarded as a potentially stale capture at the time of processing.

Event sourcing inherently provides fault tolerance capabilities when the event logs are persisted in a reliable way. Messages and vertex states are stored and can be directly reconstructed after a crash. When using a fault-tolerant and persistent messaging service (e.g., Apache Kafka), fault tolerance can be added to a platform implementing the CHRONOGRAPH concept with manageable overhead. After a crash, a worker restores the latest states of its vertices, checks and potentially resends outgoing in-transit messages, and starts consuming unprocessed inbound messages buffered by the messaging service.

9 CONCLUSIONS

In this paper, we presented CHRONOGRAPH, a distributed platform concept for computations on event-driven, dynamically evolving graphs. The platform combines a vertex-centric execution model with a distributed event sourcing mechanism for graph data. The platform is able to consume external event streams and run message-driven graph computations in a highly asynchronous and parallel manner. At the same time, the platform can retrieve causally consistent snapshots of the graph from the event logs of the vertices. These snapshots are decoupled from the live graph and form the basis for other programming and processing models that require more synchronicity, strict immutability of the topology, or long-running operations. Also the dynamicity and history of the graph can be subject of computations — either by iteratively processing consecutive snapshots, or by processing the event logs of single vertices. Unlike other platforms, we take into account the evolution and full persistence of the graph and natively support interactions of the graph platform with external systems as part of the application logic. Our performance measurements confirmed the general feasibility of the platform concept, but also revealed some room for performance improvements for the platform implementations. In summary, CHRONOGRAPH enables graph-based, data-intensive applications to apply a number of different programming and processing models within a single platform.

ACKNOWLEDGMENTS

This work has been supported by arago GmbH, Germany. The authors alone are responsible for the content of this paper.

REFERENCES

[1] 2015. Introducing Gelly: Graph Processing with Apache Flink. http://flink.apache.org/news/2015/08/24/introducing-flink-gelly.html. (2015). Accessed: 2017-03-02.
[2] Gul Agha. 1986. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, USA.
[3] Sergey Brin and Lawrence Page. 1998. The anatomy of a large-scale hypertextual web search engine. *Computer networks and ISDN systems* 30, 1 (1998), 107–117.
[4] Andrei Broder, Ravi Kumar, Farzin Maghoul, Prabhakar Raghavan, Sridhar Rajagopalan, Raymie Stata, Andrew Tomkins, and Janet Wiener. 2000. Graph structure in the web. *Computer networks* 33, 1 (2000), 309–320.
[5] Jonathan Cohen. 2009. Graph twiddling in a mapreduce world. *Computing in Science & Engineering* 11, 4 (2009), 29–41.
[6] Gianpaolo Cugola and Alessandro Margara. 2012. Processing Flows of Information: From Data Stream to Complex Event Processing. *ACM Comput. Surv.* 44, 3, Article 15 (June 2012), 62 pages.
[7] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.
[8] Benjamin Erb and Frank Kargl. 2015. A Conceptual Model for Event-sourced Graph Computing. In *Proc. of the 9th ACM International Conference on Distributed Event-Based Systems.* ACM, New York, NY, USA, 352–355.
[9] Benjamin Erb, Dominik Meißner, Gerhard Habiger, Jakob Pietron, and Frank Kargl. 2017. Consistent retrospective snapshots in distributed event-sourced systems. In *Proc. of the International Conference on Networked Systems 2017.* 1–8.
[10] Raul Castro Fernandez, Peter Pietzuch, Jay Kreps, Neha Narkhede, Jun Rao, Joel Koshy, Dong Lin, Chris Riccomini, and Guozhang Wang. 2015. Liquid: Unifying Nearline and Offline Big Data Integration. In *Proc. of the 7th Biennial Conference on Innovative Data Systems Research.*
[11] M. Fowler. 2005. Event Sourcing. http://martinfowler.com/eaaDev/EventSourcing.html. (2005). Accessed: 2017-02-14.
[12] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation.* USENIX, Hollywood, CA, 17–30.
[13] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework. In *Proc. of the 11th USENIX Conference on Operating Systems Design and Implementation.* USENIX Association, Berkeley, CA, USA, 599–613.
[14] Anand Padmanabha Iyer, Li Erran Li, Tathagata Das, and Ion Stoica. 2016. Time-evolving graph processing at scale. In *Proc. of the 4th International Workshop on Graph Data Management Experiences and Systems.* ACM, 5.
[15] Vasiliki Kalavri, Vladimir Vlassov, and Seif Haridi. 2016. High-Level Programming Abstractions for Distributed Graph Processing. *arXiv preprint arXiv:1607.02646* (2016).
[16] Martin Kleppmann. 2017. *Designing Data-Intensive Applications.* O'Reilly.
[17] Ten H Lai and Tao H Yang. 1987. On distributed snapshots. *Inform. Process. Lett.* 25, 3 (1987), 153–158.
[18] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A System for Large-scale Graph Processing. In *Proc. of the 2010 ACM SIGMOD International Conference on Management of Data.* ACM, New York, NY, USA, 135–146.
[19] Nathan Marz and James Warren. 2013. *Big Data: Principles and best practices of scalable realtime data systems.* Manning Publications.
[20] Frank McSherry, Michael Isard, and Derek G. Murray. 2015. Scalability! But at what COST?. In *15th Workshop on Hot Topics in Operating Systems.* USENIX Association, Kartause Ittingen, Switzerland.
[21] Youshan Miao, Wentao Han, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Enhong Chen, and Wenguang Chen. 2015. ImmortalGraph: A System for Storage and Analysis of Temporal Graphs. *Trans. Storage* 11, 3, Article 14 (July 2015), 34 pages.
[22] Alan Mislove, Massimiliano Marcon, Krishna P. Gummadi, Peter Druschel, and Bobby Bhattacharjee. 2007. Measurement and Analysis of Online Social Networks. In *Proc. of the 7th ACM SIGCOMM Conference on Internet Measurement.* ACM, New York, NY, USA, 29–42.
[23] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *Proc. of the 24th ACM Symposium on Operating Systems Principles.* ACM, 439–455.
[24] Red Bull Media House and Martin Krasser. 2015. Eventuate Toolkit. http://rbmhtechnology.github.io/eventuate/. (2015). Accessed: 2017-03-03.
[25] K. Sankaralingam, S. Sethumadhavan, and J. C. Browne. 2003. Distributed pagerank for P2P systems. In *High Performance Distributed Computing, 2003. Proceedings. 12th IEEE International Symposium on.* 58–68.
[26] Yogesh Simmhan, Charith Wickramaarachchi, Alok Gautam Kumbhare, Marc Frîncu, Soonil Nagarkar, Santosh Ravi, Cauligi S. Raghavendra, and Viktor K. Prasanna. 2014. Scalable Analytics over Distributed Time-series Graphs using GoFFish. *CoRR* abs/1406.5975 (2014).
[27] Greg Young. 2013. CQRS Documents. https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf. (2013). Accessed: 2017-02-14.