GraphTides: A Framework for Evaluating Stream-based Graph Processing Platforms

Conference Paper · June 2018

DOI: 10.1145/3210259.3210262

7 authors, including:

Benjamin Erb
Ulm University
35 PUBLICATIONS **140** CITATIONS
SEE PROFILE

Dominik Meißner
Ulm University
9 PUBLICATIONS **24** CITATIONS
SEE PROFILE

Benjamin Steer
Queen Mary, University of London
8 PUBLICATIONS **20** CITATIONS
SEE PROFILE

GraphTides: A Framework for Evaluating Stream-based Graph Processing Platforms

Benjamin Erb
Dominik Meißner
Frank Kargl
[firstname].[lastname]@uni-ulm.de
Institute of Distributed Systems
Ulm University, Germany

Benjamin A. Steer
Felix Cuadrado
b.a.steer@qmul.ac.uk
felix.cuadrado@qmul.ac.uk
Queen Mary, University of London
London, United Kingdom

Domagoj Margan
Peter Pietzuch
d.margan15@imperial.ac.uk
prp@doc.ic.ac.uk
Imperial College London
London, United Kingdom

ABSTRACT

Stream-based graph systems continuously ingest graph-changing events via an established input stream, performing the required computation on the corresponding graph. While there are various benchmarking and evaluation approaches for traditional, batch-oriented graph processing systems, there are no common procedures for evaluating stream-based graph systems. We, therefore, present GraphTides, a generic framework which includes the definition of an appropriate system model, an exploration of the parameter space, suitable workloads, and computations required for evaluating such systems. Furthermore, we propose a methodology and provide an architecture for running experimental evaluations. With our framework, we hope to systematically support system development, performance measurements, engineering, and comparisons of stream-based graph systems.

CCS CONCEPTS

• **General and reference** → **Evaluation**; **Measurement**; • **Information systems** → *Data streaming*; • **Theory of computation** → *Graph algorithms analysis*;

KEYWORDS

graph processing; graph analytics; stream-based graphs; evolving graphs; temporal graphs; evaluation; measurements

ACM Reference Format:
Benjamin Erb, Dominik Meißner, Benjamin A. Steer, Domagoj Margan, Frank Kargl, Felix Cuadrado, and Peter Pietzuch. 2018. GraphTides: A Framework for Evaluating Stream-based Graph Processing Platforms. In *GRADES-NDA'18 : 1st Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA), June 10–15, 2018, Houston, TX, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3210259.3210262

GRADES-NDA'18 , June 10–15, 2018, Houston, TX, USA

ACM ISBN 978-1-4503-5695-4/18/06…$15.00
https://doi.org/10.1145/3210259.3210262

1 INTRODUCTION

Most existing graph processing platforms consider computations on static graphs (e.g., Pregel [34], PowerGraph [18], GraphX [19]). These platforms take an arbitrary, static graph as input, run a specified graph computation — in most cases in a batch-oriented fashion — and eventually yield a resulting graph or other output. Dynamic graphs with mutating topologies have also been addressed by some platforms (e.g., Chronos [22], GoFFish [44]). As most real-world graphs such as social networks or web graphs change over time, their dynamicity may be captured, often by periodically creating snapshots. Graph snapshots of different points in time are then processed in batches to perform temporal graph computation. An even smaller subset of systems address the challenge of dynamic graph processing by using streams of changes (Kineograph [9], GraphTau [27], Chronograph [15], GreyCat [23], KickStarter [45]).

Stream-based graph processing systems perform graph computations while simultaneously receiving a stream of graph updates. As the actual graph is still evolving during runtime, these system can apply a spectrum of different computation styles, ranging from batch computations on internal snapshots to online computations. While the former can provide more exact, but potentially stale results, the latter provides fast, but often approximate results. This establishes a trade-off between correctness and latency.

Figure 1 shows an example of a stream-based graph system that is fueled by update operations from a social network and that manages an evolving graph model (as up-to-date as possible) while at the same time performing approximative computations on the graph. Approximation allows tasks such as identifying influential users and trending content to complete within a timely manner. This is important, as longer computations with a higher accuracy would be based on a staler input when completed, or results would not be available in time. Stream-based graph systems sit between batch graph analytics systems and graph database stores. Their workload is more interactive as with traditional graph analytics systems, and their computations go beyond graph queries and traversals. On the other hand, stream-based graph systems do not necessarily provide any persistence or advanced query support, nor are their computation results expected to always be accurate.

The popularity and relevance of traditional, batch-oriented graph processing systems has led to numerous efforts to systematically benchmark and evaluate such systems [2, 7, 21, 26, 39], as shown in section 2. We argue that evaluation techniques are just as important for stream-based graph processing systems. They (i) facilitate design and implementation decisions during development, (ii) provide

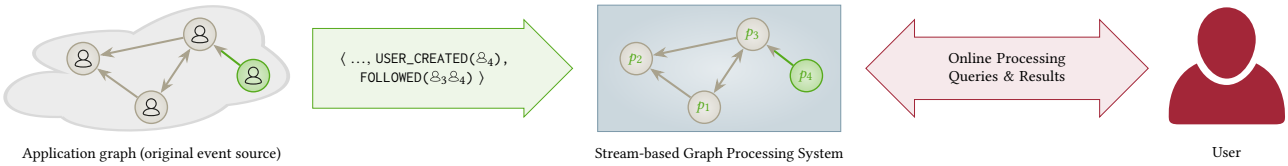


Figure 1: General model of stream-based graph processing systems. An event source provides a persistent stream of graph update events. The system ingests the stream, maintains an internal graph representation, and executes graph computations.

reliable performance estimations, (iii) enable targeted performance engineering steps, and (iv) allow for unbiased system comparisons. The increasing relevance of near-realtime graph processing necessitates a thorough exploration of its design spaces and evaluation concepts, presented in section 3, which go beyond existing solutions and consider perspectives from domains, such as event processing.

As a main contribution of this work, we introduce a generic framework for evaluating stream-based graph processing systems in section 4, complemented by a methodology and an architecture for performing tests and measuring system performance. A prototype implementation of this architecture is then used for the experimental evaluation of two existing systems in section 5. Finally, in section 6, we conclude our contributions and point to potential future work.

2 BACKGROUND

Our approach relates to existing evaluation works carried out for conventional graph-based and stream-based systems. We also survey existing works on system evaluation methodologies and point out typical use cases for stream-based graph processing.

2.1 Evaluation of Graph-based Systems

Prior work for the evaluation of graph-based systems focused primarily on the following: (i) traditional, batch-oriented graph processing systems and (ii) graph database systems. We do not go into details for benchmarks for specialized semantic graph systems (e.g., Berlin SPARQL Benchmark [6], LUBM [20]), or graph systems for high performance graph analytics (e.g., Graph 500 [12]), HPC Scalable Graph Analysis Benchmark [5]) , as they do not fit into our focus of general purpose, community-driven graph processing systems for single machines and commodity clusters.

LDBC Graphalytics [26] is currently the only industrial-grade benchmark designed and developed for the evaluation of graph processing systems. It supplies out-of-the-box benchmark suites for existing graph processing systems both from industry and the open source community, but also supports the custom evaluation of arbitrary graph processing platforms. Graphalytics offers a benchmarking palette of six standard graph algorithms and a number of synthetic and real datasets, provided in four different size classes. When not relying on real data sets, graph benchmarks require synthetic graphs. Graph generators are still an active field of research [8, 41, 42]. However, not all generators provide results that can be streamed. The most basic benchmark provided by Graphalytics consists of measuring the baseline performance in terms of total processing time and processed vertices/edges per second, all of which are obtained by running a variety of experiments with different datasets and algorithms. Different systems may then be compared based on quantifying metrics for scalability, robustness, and performance variability. Graphalytics provides an additional set of tools as part of the suite. The LDBC SNB generator [16] creates synthetic workloads based on a social network graph. Granula [39] is a performance evaluation tool of Graphalytics used for additional fine-grained, manual bottleneck analysis in graph processing systems. While Graphalytics represents an entire benchmarking suite for batch-oriented graph analytics, our framework addresses the evaluation of stream-based graph processing systems not covered by Graphalytics. McSherry et al. [36] pointed the relevance of rigorous evaluations of graph processing systems in terms of scalability and performance properties.

Graph databases specialize in efficient storage, querying, and manipulation of the entities and relationships which form a graph structure. XGDBench [13], LinkBench [4], or the Graph Traversal Benchmark [11] are examples for benchmarks with graph database workloads that primarily target traversal-based queries. The necessity of dedicated workloads that address real-time social networking has been discussed in [40]. Although the authors limit their discussion on graph databases and relational stores, their general approach shares some similarities with our evaluation framework for stream-based graph processing.

2.2 Evaluation of Stream Processing Systems

While benchmark suites for graph analytics primarily focus on overall performance, scalability, and duration of computations, systems for stream processing and event processing have slightly different sets of metrics. As these systems often aim for near-realtime processing and results with low latency, their behavior over time and under varying load becomes more relevant. This is reflected in various benchmarks for stream processing systems such as HiBench [24], SparkBench [32], BigBench [17], and BigDataBench [46].

Prior to the existence of stream processing benchmarks, developers of stream processing systems had to perform their own experiments to evaluate and compare the performance of their systems. Initial attempts for evaluation and comparison of emerging stream processing systems were found within existing batch benchmarking suites which extended their capabilities to support streaming scenarios. The earliest example of a stream management benchmark is LinearRoad [3], which aimed at the evaluation of pioneering stream data management systems such as Aurora [1] and STREAM [38]. StreamBench [33] is one of the first benchmarking suites exclusively designed and developed with modern distributed stream processing systems in mind (namely Spark and Storm). It offers seven different streaming workload scenarios, performance metrics for different data scales and the evaluation of fault tolerance capabilities. Recent stream benchmarking efforts such as the Yahoo! benchmarking

suite [10] focus on measuring latency produced by popular stream processing engines (e.g., Storm, Spark, and Flink) under a given input load.

The usage of stream-based, replayed input and the focus on runtime behavior under load is something that we adopt for our evaluation approach for stream-based graph processing systems.

2.3 Evaluation Methodologies

Performance evaluations have a long history in the systems community for performance engineering of system components during development, for analyses of full systems and for comparison between alternative solutions. Jain [28] has established a rigorous performance evaluation methodology for computer systems. Firstly, the system and its components have to be defined and the goals of the analysis have to be stated. The function of the system and its desired outcomes must then be listed. Next, the analyst must choose appropriate metrics, including the type of metric (e.g., throughput, response time) and the optimum value in the chosen dimensions (e.g., *HB: higher is better*, *LB: lower is better*, *NB: nominal is best*). Following this, a list of parameters affecting the performance of the system must be drawn up — both system parameters and properties of the actual workload. The analyst can then decide which parameters are to be varied for the experiment, and define appropriate levels for those remaining. The evaluation can either be implemented as a simulation, by providing an analytical model, or by measuring a real system. The workload description characterizes the service usage under test. For the experimental design, the analyst chooses a number of setups. This can range from variations of a single parameter, to full factorial designs where all levels of all factors are considered. In the case of benchmarks, different systems are measured with the exact same setup. Finally, after analysis and interpretation of the data, results can be presented.

Popper [29] represents a modern approach for conducting systems experiments which take into account automation and reproducibility of experiments. The Popper pipeline defines a workflow between source code management, packaging, input data management, execution, data and metrics collection, analysis, and finally publication of results. It also specifies a skeleton structure for experiment repositories and a number of pre-defined scripts which interact with the Popper command line tool. The Popper convention can be implemented using best practices from software engineering and software operation (DevOps). For instance, solutions for lightweight OS-level virtualization (e.g., Docker) can be used for packaging an experiment. Published experiments which follow the Popper convention are intended to be easily re-executable by others because of this established procedure.

2.4 Use Cases

We illustrate three example use cases that represent typical application areas of evolving graphs and stream-based graph systems.

Connections in a Social Network. A social network grows as more users sign up and steadily connect with friends. A stream-based graph system processes each change in the social graph and maintains a ranking value for each user indicating their influence. The system also uses the changes to detect certain trends in the social graphs, such as individuals that attract a lot of new friends within a specified period or communities that slowly break apart due to users leaving the network.

Distributed Denial-of-Service Attacks in Computer Networks. A stream-based graph system supervises a set of servers in a computer network, modeling traffic flow between the servers and remote clients. The system receives streams of flow data from the ingress router and software-defined network switches within the monitored network. While the traffic of individual attackers may appear benign to the servers, the full insight of all incoming streams allows the stream-based graph system to identify anomalous temporal traffic patterns in the network and to determine the attackers' networks. Malicious hosts and remote networks may then be blacklisted.

Blockchain Transactions. In a distributed ledger, new blocks represent micro-batches of transactions. Each transaction contains the parties involved and the transaction amounts. A stream-based graph processing system consumes the stream of transactions and maintains a combined transaction/wallet graph. The system provides live statistics for the transaction network and specific wallets. These include balances, average transaction values, and distribution of holdings in the network over time.

3 TOWARDS THE EVALUATION OF STREAM-BASED GRAPH SYSTEMS

Before introducing our framework, we discuss the implications of evaluating stream based graph processing systems. We, therefore, first describe a general model for the stream-based graph systems we are taking into account. Next, we discuss the altered parameter space due to the dynamicity of a stream-based perspective. Finally, we list the general requirements for the evaluation framework.

3.1 System Model

In our model, stream-based graph processing platforms are computing systems which receive a continuous, unbounded stream of events from an external source via the network (see Figure 1). The event source is a remote system acting as a proxy for an actual real-world graph. The graph is both stateful and mutable. The source publishes changes from its graph as soon as they become observable. Each event in the stream contains a single, localized operation which describes a change within the graph topology (i.e., by adding or removing a vertex or edge) or an update to its state (i.e., by updating the properties of a vertex or edge). Therefore, the ordered stream $S_{G,t}$, which contains the events s_1 up to s_t, defines a corresponding graph $G_t = (V, E)$ at time t:

$$S_{G,t} = \langle s_1, s_2, \ldots s_t \rangle \Leftrightarrow G_t = (V, E)$$

Each event s_n describes one of the following six operations: add/remove vertex/edge; update vertex/edge state. The sets of vertices V and edges E and their corresponding states are thus defined by the sequence of all previously executed graph operations. The graph system processes each incoming event from the stream and updates its own internal representation of the graph correspondingly.

A main goal of an online platform is to keep the internal graph representation as up-to-date as possible; otherwise, this representation would reflect a stale version of the actual graph. Note, the event source and the processing platform are not coupled and their

event rates might differ. The systems use the graph representation to execute online graph computations, therefore, graph computations and graph mutations run concurrently. Computation results can be queried by the user once available. Note that this approach differs from graph stream algorithms [35] which directly compute pre-defined metrics without maintaining a graph representation.

3.2 Parameter Space

Benchmarks for batch-oriented graph processing systems focus primarily on performance metrics as a function of the input graph, the selected graph algorithm, and of the system under test.

When considering graph processing with streams of graph updates, the space of relevant parameters and metrics expands significantly. This is particularly noticeable for online processing, when timeliness and low latency become even more relevant. We now discuss all parameters and argue which we restrict *ab initio* and to what extent others can be varied.

Graph Types. In our model, graphs are directed and stateful. Both vertices and edges possess a mutable state. We do not, however, consider multigraphs or self loops. Vertices are identified by a unique ID. Undirected graphs can be modeled by ignoring edge directions and stateless graphs can be represented by ignoring state properties. This model is sufficient enough to cover most domains of dynamic and evolving graphs such as social networks, computer networks or road traffic networks.

Graph Evolution Properties. Probably the most striking difference between stream-based graphs and static input graphs is their dynamicity. A static graph possesses a number of structural graph properties, including the number of vertices and edges, degree distributions, or measures for distance, connectivity, and centrality.

The dynamic aspect of evolving graphs adds additional temporal graph properties [31, 43, 47]. Not only can all previous structural graph properties change over time, but the dynamicity is also reflected in the rate, locality and distribution of change events — both for topology changes (i.e., churn rate of vertices and edges) and for state updates of vertices and edges. Different streams can lead to distinct temporal workload patterns, e.g., highly concurrent topology changes at various points in the graph compared to huge numbers of state update operations on a single vertex.

By allowing different streams of change operations, the graph can evolve arbitrarily. In fact, one challenge of creating such workloads is the preservation of certain graph properties.

Streaming Properties. With regards to the stream of updates, ordering, reliability, and messaging semantics have to be considered. We argue that strong guarantees by default are expected by many online graph processing platforms. Altered event orders or the loss of events may produce inconsistent graph topologies, as operations might fail due to violated preconditions caused by lost preceding events. The framework is expected to constantly run evaluations based on an ordered and reliable stream of events which follow exactly-once semantics. However, by appropriately altering the input stream used for an evaluation run, the framework can deterministically replay streams which do not adhere to the aforementioned guarantees. While it is straightforward to modify a reliable, ordered stream into an unreliable, unordered stream (e.g., by dropping or duplicating arbitrary events or by shuffling partial streams), the opposite conversion is not trivial. This is why we require strong semantics for the stream replay, but allow the analyst to inject faults into the input stream *a priori* (i.e., reordering, duplicates, losses).

Another parameter of stream processing is the extent to which systems can apply backpressure. Pull-based streaming APIs often allow systems to throttle down the rate of events sent. Alternatively, the flow control mechanism of TCP can be used to indicate overload. Without backpressuring, the system has to provide its own mechanism to prevent losing events under load. Systems either rely on internal mechanisms to handle bulk loads (e.g., scale up, scale out), or they have to offload pending events to an external component (e.g., a scalable, persistent message queue).

Concurrency & Parallelism. A single, ordered input stream emitted by multiple event sources requires constant coordination between the sources to retain ordering. Coordination, in turn, reduces the streaming performance and can only mitigate the occurrence of violated guarantees, not remove them entirely. As a result, a stream is only allowed to have a single event source in our model.

In order to enable parallelism and horizontal scaling of input workload, we opt for concurrent streaming of disjunct streams by different event sources; multiple independent graphs are provided and changed concurrently. We argue that the resulting processing load is similar to the load of a single graph with concurrent changes for many computations.

3.3 Requirements

Our framework addresses four main features required for the evaluation of stream-based graph processing systems.

Platform-agnostic framework. To date, different platforms for online graph processing have emerged with substantially different concepts in terms of their programming model, processing style, and streaming semantics. Hence, the suggested evaluation framework requires a platform-agnostic design in order to support current and future stream-based graph systems. This includes a generic streaming interface supporting different modes of operation (i.e., push-based and pull-based), which can be adapted by platform-specific connectors. Instead of specifying concrete algorithm implementations, the framework should list computation goals to be provided by the analyst for the system under test.

Openness & extensibility. As previously outlined, the parameter space for stream-based graph processing exceeds the complexity of existing batch-oriented approaches, hence, the framework should be modular and easy to extend. In particular, new algorithms for generating graph streams should be able to be added to reflect different evolving graphs. Existing graph streams — synthetically generated by an external tool or captured from real graphs — should also be supported as input.

Representative and versatile workloads. Regarding data sets, processing tasks, and metrics, we need to take into account stream-specific workloads resembling computations typically executed on evolving graphs. This includes the generation of fast, approximative results and corresponding metrics for assessing latency vs. correctness trade-offs, but also continuous, iterative queries.

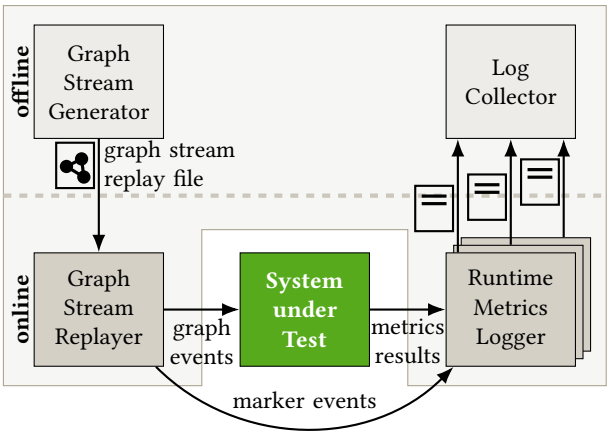


Figure 2: Conceptual overview of the GraphTides framework and test harness.

Structured evaluation process. A single test run with an arbitrary workload does not represent a valid evaluation. Hence, the framework should define specific steps on how to run evaluations. This includes different evaluation goals (e.g., increasing the input stream rate), relevant evaluation factors and performance metrics, and basic considerations for analysis and interpretation of results.

4 THE EVALUATION FRAMEWORK

Our evaluation framework GraphTides comprises metrics, workloads, components of the evaluation architecture, and methodological recommendations. An implementation of GraphTides is available under an open source license[1]. Figure 2 illustrates the general design of the framework. The test harness induces a stream of graph events onto the system under test. At the same time, the test harness continuously gathers metrics and processing results from the system.

In order to support the evaluation of different systems with varying levels of internal access, we defined three *evaluation levels*.

- **Level 0** A system under test on this level is considered a black box. The system provides an interface for ingesting the graph stream and for accessing or querying computation results.
- **Level 1** Systems on level 1 extend level 0 in such a way that they expose a native metrics interface. This additional source of information can be used to collect platform-internal data at runtime (e.g., current throughput, platform load), as provided by the system.
- **Level 2** On this level, a system provides complete access to its internals and the analyst is able to make modifications to its source code. Specific measurement logic can be injected into the system. This evaluation level aligns with the fine-grained perspective of Granula [39].

Concrete evaluation steps depend on the maximum level supported by the system, but also on the the actual evaluation goal. For instance, the comparison of two stream-based graph processing systems in regard to their average load is possible on level 0. In-depth performance engineering of a system's internal scheduling component requires an evaluation on level 2.

[1] https://graphtides.github.io

4.1 Framework Architecture

GraphTides consists of a *graph stream generator* for the creation of graph streams, a test harness, and a component for collecting results (see Figure 2). During a test run, the *graph stream replayer* outputs a generated stream, feeding the system under test while *runtime metrics loggers* gather continuous data from the system, its processing results, and other sources. After a finished run, the *log collector* aggregates the recorded data and yields a single result log.

The graph stream file contains a sequence of entries of the following types: (i) graph-changing events, (ii) marker events for flagging specific points in time in the stream, and (iii) events that control the behavior of the replayer. The graph stream input is typically split into two parts, divided by a marker and a pause event. The first phase bootstraps the initial graph and warms up the system under test, while the second represents the main evaluation phase.

In order to connect the test harness to the system under test, the analyst either plugs a platform-specific connector into the *graph stream replayer* component, or provides logic within the platform to receive and terminate the inbound stream accordingly.

Although all components, including the system under test, could be placed on the same machine, we recommend a distributed setup that conforms with typical use cases: external event sources, network-based streams, and remote access to processing results. Experimental setups can also be scaled by adding multiple replayer instances (each providing separate streams), and multiple system instances (in case the system under test supports distribution).

Measurements in our framework are timestamped, requiring synchronized wall clocks for all components of the framework whenever measurements from different machines are examined as a whole (e.g., by using the Precision Time Protocol [25]).

4.2 Graph Stream Format

The graph stream is a plain comma-separated value file which contains a single event per line: `COMMAND`, `ENTITY_ID`, `PAYLOAD`. Graph-changing events add or remove a vertex/edge or update its state. Vertices are identified by a unique ID and edges are identified by concatenating the source and destination identifiers, separated by a dash. States of vertices and edges are user-defined strings (e.g., stringified JSON). Marker events are particularly important for pointing to specific stream segments. In a later analysis, time-specific metrics can be correlated with the marker event. Markers can also signal that a certain input has (not) entered the system and that a corresponding computation result should now be expected. Control events can change the speed of the replayer at runtime by defining a speed-up factor (1 represents the initially defined rate of the replayer). This allows emulation of varying rates, and is helpful for inducing short bursts and peaks. A second control event causes the replayer to pause new events for a specified amount of time.

4.3 Metrics

For online processing systems, the behavior over time and under different workloads is of particular interest. This is why we focus on time-series values for most metrics, collected at runtime by a set of logger instances.

For some metrics, we also consider aggregated values when directly comparing different solutions. This includes metrics such

as average event throughput (HB), 99th percentile or result latency (LB), or median relative errors of approximation results (LB).

Level 0 Metrics. Without any insights into the system, the analyst relies on agnostic profiling tools to periodically measure the graph system processes (e.g., `perf`, `pidstat` for Linux-based systems) and the underlying operation system (e.g., `dstat`, `iostat`, `sar` for Linux-based systems). For each process, CPU load (LB), memory usage (LB), network I/O (LB), and disk I/O (LB) have to be logged[2]. Measuring these metrics on a per-machine level also facilitates the identification of confounding background processes.

Level 1 & 2 Metrics. Additional internal metrics, either provided by a monitoring API or by a system-internal hook, are useful for identifying specific load patterns, choke points, and bottlenecks. Among others, these metrics could include internal throughput rates (HB) and communication latencies between workers (LB).

Streaming Metrics. The graph stream replayer or system client library is instrumented to collect metrics on the actual ingress rate (HB) and for correlating in-stream marker events with wall-clock time stamps in later analyses.

Computation Metrics. Metrics for assessing computation results depend on the actual computation. For most results, this is the *latency* (LB) until the result is available, and then its *accuracy* (HB). Latency describes the time it takes for an ingested input event to be reflected in a computation result. Accuracy describes the extent of discrepancy between the result provided by the platform and the exact result. Exact results are part of the workload definition and need to be prespecified (i.e., by reconstructing the target graph and running a separate batch computation as reference). *Correctness* is an alternative metric for assessing computations that yield a dichotomous result (i.e., existence of a path at a certain time).

4.4 Workloads

A workload in our framework is defined by at least one graph stream and any number of computations to be executed during streaming.

4.4.1 Graph Stream Properties. A graph stream is defined as a sequence of events to be replayed. It characterizes the load induced on the platform in terms of graph dynamicity over time and possesses the following properties:

Stream Composition Sequence of topology-changing events and events which update the state of vertices and edges.
- *Event mix*: Ratio of topology changing events vs. state changing events.
- *Interleaving*: Patterns of how both types alternate in the stream.

Topology changes Growth/decay of the graph over time.
- *Direction*: Ratio of add vs. remove operations.
- *Types*: Ratio of vertex vs. edge modifications.
- *Locality distribution*: Positions of the vertices/edges to be added/removed.
- *Temporal distribution*: Patterns of topology changes over time.

State changes Updates of vertex and edge values.
- *Types*: Ratio of vertex vs. edge updates
- *Locality distribution*: Positions of the vertices/edges to be updated.
- *Temporal distribution*: Temporal patterns of state updates.
- *Update behavior*: Workload-specific characteristics of how states evolve.

Streaming rate The *stream rate variability* is defined by the graph stream file which controls the streaming rate during a replay.

4.4.2 Computations. Potential computations on stream-based graphs depend on the programming model and the processing style of the system, but they also differ in the semantics of their computations. Offline computations are executed on graph snapshots that are reconstructed from the event stream (e.g., epoch snapshots in Kineograph [9]). Online computations directly process incoming graph stream events (e.g., live model of Chronograph [15]). Hybrid approaches (e.g., pause/shift/resume in GraphTau [27]) combine both approaches. Converging computations (e.g., online PageRank variants, distributed routing algorithms) can be executed on evolving graphs. The accuracy of their results at any instant of time is then shaped by the duration of the preceding computation, and the extent of recent changes. Other algorithms always yield a definite result (e.g., triangle count). In online computations on evolving graphs, this result may be inaccurate once provided, as it is based on a stale view of the graph. Note that this is still a valid output, as it provides a useful estimation. The periodical execution of computations can also yield time-series data on relevant properties (e.g., diameter estimation).

The choice of computation depends on the evaluation goal and the capabilities of the system. Table 1 depicts a sample overview of potential computations for many workloads.

During an evaluation run, the test harness continuously queries the platform for results or updates to previous results of the computation. This is particularly relevant for online computations that provide result approximations. Combined with the marker events in the input stream, this allows for an estimation of the result accuracy and timeliness of computations.

4.5 Evaluation Methodology

For the evaluation methodology, we suggest a combination of Jain's methodology [28] and the Popper conventions [29] for automation and reproducibility. Specific evaluations are first defined by the goals ranging from rough performance tests to specific optimizations of a single platform component. Appropriate workloads and metrics are then derived from these goals. Performance measurements consist of a gradual exploration of the experiment factors/levels and the corresponding impact on the outcome variables. Experiments during system development are primary used in iterative steps where the same workload is applied to a gradually modified system. The assessment of results is grounded on an analysis of the combined log file of an experiment run. Tools for the assessment include appropriate visualizations (e.g., time series plots) and statistical time series analyses (e.g., cross-correlations).

For evaluative comparisons between multiple systems or internal component alternatives, we advise statistically rigorous testing.

[2]Optima depend on the characteristics of the system. For a CPU-bound system, increased network load may be desirable if the overall performance increases.

Table 1: Example computations for stream-based graph systems.

Computation	Examples
Graph statistics	Global properties, degree distribution
Graph properties	PageRank, cycle detection
Routing & traversals	Bellman-Ford, Floyd-Warshall, breadth-first search, spanning tree construction, diameter estimation
Graph theory	Vertex coloring, triangle count
Communities	Weakly connected components, k-means, community detection
Temporal analyses	Trend analyses on graph properties, online sampling

In particular, this requires at least $n \geq 30$ test runs for each configuration due to the central limit theory. Results can then be compared using confidence intervals of the aggregated metrics (often CI_{95}). Non-overlapping confidence intervals of the results from two different systems are indeed significantly different under the given interval. Although this approach requires more effort, it minimizes the risks of wrong conclusions.

An important evaluation pattern that we have encountered during our tests are *watermark events* for temporal correlations in the result logs. Pre-defined marker events within the stream are emitted to a logger and indicate that the preceding event has been streamed into the system. This is particularly helpful when assessing the time it takes until a change is either reflected in the internal graph, or in the corresponding computation result.

5 EXPERIMENTAL EVALUATIONS

In order to evaluate our framework, we implementated and tested its components, benchmarked the graph stream replayer, and briefly applied the methodology to two existing stream-based graph systems. We followed the Popper conventions [29] for evaluations and all steps were automated with a set of shell scripts. The systems under tests were packaged as Docker containers. The distributed evaluation setup was realized with Docker Swarm. The scripts are part of the examples available in the GraphTides repository.

5.1 Implementation of GraphTides Prototype

The *graph stream generator* is implemented in TypeScript (a statically typed superset of JavaScript) using Node.js. Graph generation rules can be specified by a set of user-defined functions (see Listing 1). The graph stream generation is conceptually divided in two phases: (i) bootstrapping an initial graph, and (ii) continuously modifying the resulting graph. In practice, this split makes it easier to use a well-known graph generation algorithm for the initial graph (such as Barabási-Albert or Erdős-Rényi) and to provide a more specific algorithm for the subsequent evolution phase. The generator works in a configurable number of rounds. In each round, a user-defined function selects the event type of the round and an appropriate target vertex/edge. A user-defined callback additionally allows the modification of the state of the target vertex/edge. The *graph stream replayer* is implemented in Java 9 and is specifically designed for emitting a stream of events with a uniform, yet tunable event rate. Therefore, streaming is decoupled from reading the stream graph file. We use a multi-threaded design to decouple both tasks and to ensure high throughput. Emitting stream events is handled by a dedicated thread that uses high precision timestamps and busy-waiting for timeliness. For our evaluations, we used small Python and Node.js scripts as *runtime metrics loggers*. These scripts periodically executed a certain operation such as submitting a certain query, collecting a certain metric, or receiving a marker event. Each logger appends a time stamp to any outcomes and writes them into a local log. Once a test run is finished, the *log collector* script gathers the remote log files of all logger instances and merges them into a single, chronologically sorted result log file.

5.2 Evaluation of Graph Stream Replayer

The stream replayer represents the critical component for test performance. Our implementation is able to achieve robust streaming rates even with a single streamer instance, both for piped and TCP-based connections (Figure 3a). For target throughput rates beyond 100k events per second, the actual throughput did stick roughly to the targeted rate, but the measured range of rates increased notably.

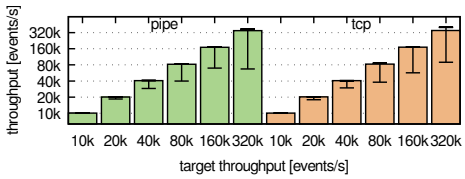
5.3 Experimental Evaluations of Real Systems

We surveyed existing stream-based graph systems and selected two recently published, research-based systems based on their availability — Weaver [14], a distributed graph store, and Chronograph [15], a distributed graph processing systems which supports online processing on evolving graphs. For the evaluations, we covered different setups (*Weaver*: single instance; *Chronograph*: distributed setup), workloads (*Weaver*: generated workload; *Chronograph*: converted SNB workload), evaluation goals (*Weaver*: ingress scalability; *Chronograph*: behavior under varying stream load), and evaluation levels (*Weaver*: level 0; *Chronograph*: level 2).

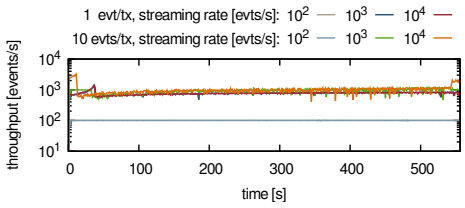
5.3.1 Write-Throughput in Weaver. Although not a graph processing system, we decided to run an experimental evaluation with Weaver, as it is designed for evolving graph-structured data with high throughput of updates [14]. Weaver provides horizontal scalability and employs transactional semantics for updates.

We used Weaver as a Level 0 example and conducted an experiment to assess Weaver's capability of write-only workloads. We configured the GraphTides replayer to forward the stream to a client process local to the Weaver instance. We then measured the actual write throughput with different target rates and transaction batches (i.e., single transaction per event vs. 10 events batched as 1 transaction) against a single instance of Weaver. As shown in Figure 3b, Weaver was only able to keep pace with lower streaming rates, while it backthrottled faster rates. The evaluation of the CPU utilization (Figure 3c) highlighted that the Weaver timestamper process consumes more cycles than other Weaver processes. This finding could represent an entry point for optimizations of Weaver.

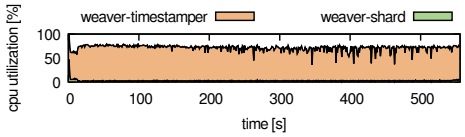
5.3.2 Chronograph. As the Chronograph prototype provides entry points to inject metrics collections, we selected it for an experimental Level 2 evaluation. We instrumented it to capture internal queue lengths and operation throughputs of the workers. We also altered it to periodically dump intermediate processing



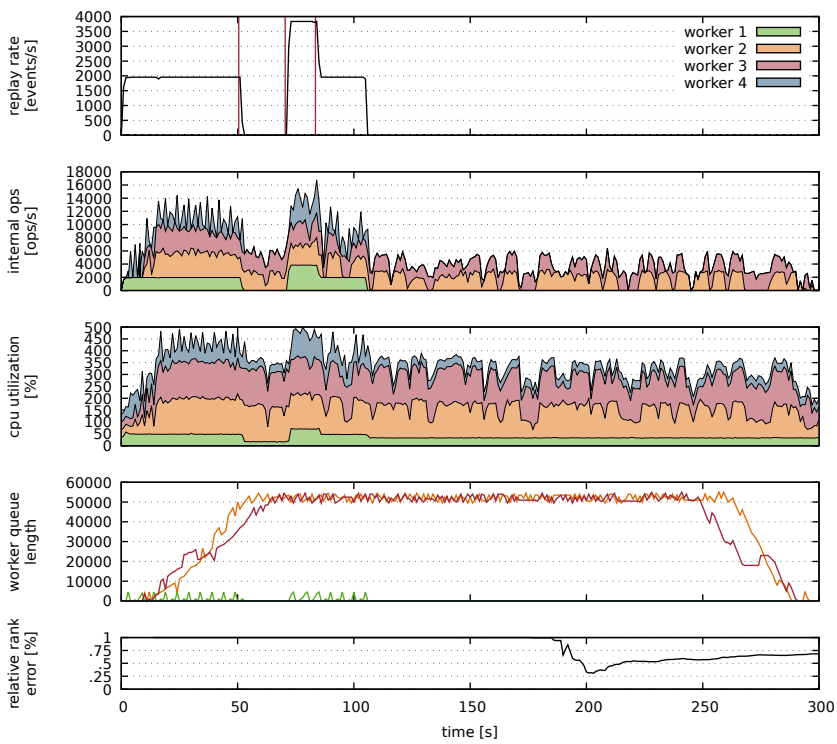
(a) Median throughput of the Graph Stream Replayer for given target rates (range covers 95%, indicates 5th percentile to maximum).



(b) Events processed in Weaver using different streaming rates and transaction batches. Independent of the actual streaming rates, Weaver appeared to have an upper bound for throughput.



(c) CPU usage of Weaver processes with 10,000 events/s badged as 10 events per transaction. The evaluation showed a relatively high utilization of the timestamper process of Weaver.



(d) Stacked time-series plot of a Chronograph experiment run with a social network workload. During graph evolution, the system computes approximate rank values for all users. Relative rank errors are estimated retrospectively. The visualization shows a long period of computation after the stream has stopped due to a large backlog of queued messages.

Figure 3: Results from our evaluation of the GraphTides framework applied to existing stream-based graph systems.

results for the most influential users. We then used a variable load which was converted from an LDBC SNB generator output. For this test, we selected a single-phase stream with a short pause of 20 seconds and a subsequent period of doubled rate. The computation executed an online rank algorithm on the evolving graph.

The experiment was executed on a setup with four workers running on four separate machines. For the assessment of the results, we generated a stack time-series plot from the result log, as shown in Figure 3d. The visualization contains data gathered from all workers as well as the instrumented replayer component and relative errors of the online computations of certain vertices. The visualization indicates that half of the worker's internal queues were saturated at the end of the stream and kept the system busy due to the backlog of internal messages for online processing. The evaluation results indicate the online algorithm in use does not align well with the Chronograph programming model, as the graph evolution and computational messages compete for internal communication resources, yielding inaccurate results with high delays.

6 CONCLUSION

Both the analysis of dynamic and evolving graphs and the general evaluation of graph processing systems continue to raise a number of research questions within the community [30, 37]. In this paper, we addressed the challenge of evaluating systems for evolving graphs by introducing GraphTides, a generic evaluation framework for stream-based graph processing systems. GraphTides tackles the timeliness dimension, which is a relevant aspect for many systems that perform stream-based graph analytics. Timeliness adds an entirely new dimension to the performance behavior and can account for changes in performance, scalability, and robustness.

GraphTides covers the full evaluation cycle from workload generation to result analysis and provides the analyst with a set of framework components to design their own experiments for a stream-based graph system. The framework supports developers of stream-based graph systems, allowing for in-depth performance measurements/engineering, and comparisons of stream-based graph systems. GraphTides might also help during the development of algorithms for online graph computations on evolving graphs. Rigorous measurements support decision-making on latency vs. accuracy/-correctness trade-offs for approximations.

Our long-term goal is to develop GraphTides into a benchmark suite — similar to LDBC Graphalytics, but for stream-based analytics. The suite needs to be generic enough to cover different computing styles on graph streams, and at the same time concrete enough for actual system comparisons. This requires future discussions within the community on standardized and representative sets of graph streams and computations as benchmarking workloads.

REFERENCES

[1] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. 2003. Aurora: a new model and architecture for data stream management. *the VLDB Journal* 12, 2 (2003), 120–139.

[2] Khaled Ammar and M Tamer Özsu. 2013. WGB: towards a universal graph benchmark. In *Workshop on Big Data Benchmarks*. Springer, 58–72.

[3] Arvind Arasu, Mitch Cherniack, Eduardo Galvez, David Maier, Anurag S Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. 2004. Linear road: a stream data management benchmark. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*. VLDB Endowment, 480–491.

[4] Timothy G. Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. 2013. LinkBench: A Database Benchmark Based on the Facebook Social Graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*. ACM, New York, NY, USA, 1185–1196. https://doi.org/10.1145/2463676.2465296

[5] David A. Bader, John Feo, John Gilbert, Jeremy Kepner, David Koester, Eugene Loh, Kamesh Madduri, Bill Mann, Theresa Meuse, and Eric Robinson. 2009. HPC Scalable Graph Analysis Benchmark. (2009).

[6] Christian Bizer and Andreas Schultz. 2009. The Berlin SPARQL Benchmark. *International Journal on Semantic Web and Information Systems (IJSWIS)* 5, 2 (2009), 1–24.

[7] Mihai Capotă, Tim Hegeman, Alexandru Iosup, Arnau Prat-Pérez, Orri Erling, and Peter Boncz. 2015. Graphalytics: A big data benchmark for graph-processing platforms. In *Proceedings of the GRADES'15*. ACM, 7.

[8] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*. SIAM, 442–446.

[9] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. 2012. Kineograph: Taking the Pulse of a Fast-changing and Connected World. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys '12)*. ACM, New York, NY, USA, 85–98. https://doi.org/10.1145/2168836.2168846

[10] Sanket Chintapalli, Derek Dagit, Bobby Evans, Reza Farivar, Thomas Graves, Mark Holderbaugh, Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, Boyang Jerry Peng, et al. 2016. Benchmarking streaming computation engines: storm, flink and spark streaming. In *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*. IEEE, 1789–1792.

[11] Marek Ciglan, Alex Averbuch, and Ladialav Hluchy. 2012. Benchmarking traversal operations over graph databases. In *Data Engineering Workshops (ICDEW), 2012 IEEE 28th International Conference on*. IEEE, 186–189.

[12] Graph 500 Steering Committee. 2017. Graph 500 Benchmarks v2.0. https://graph500.org/. (June 2017).

[13] Miyuru Dayarathna and Toyotaro Suzumura. 2012. XGDBench: A benchmarking platform for graph stores in exascale clouds. In *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on*. IEEE, 363–370.

[14] Ayush Dubey, Greg D Hill, Robert Escriva, and Emin Gün Sirer. 2016. Weaver: a high-performance, transactional graph database based on refinable timestamps. *Proceedings of the VLDB Endowment* 9, 11 (2016), 852–863.

[15] 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 the 11th ACM International Conference on Distributed and Event-based Systems (DEBS '17)*. ACM, New York, NY, USA, 78–87. https://doi.org/10.1145/3093742.3093913

[16] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. 2015. The LDBC Social Network Benchmark: Interactive Workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. ACM, New York, NY, USA, 619–630. https://doi.org/10.1145/2723372.2742786

[17] Ahmad Ghazal, Tilmann Rabl, Minqing Hu, Francois Raab, Meikel Poess, Alain Crolotte, and Hans-Arno Jacobsen. 2013. BigBench: towards an industry standard benchmark for big data analytics. In *Proceedings of the 2013 ACM SIGMOD international conference on Management of data*. ACM, 1197–1208.

[18] 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 (OSDI 12)*. USENIX, Hollywood, CA, 17–30.

[19] 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 *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, Berkeley, CA, USA, 599–613.

[20] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. 2005. LUBM: A benchmark for OWL knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web* 3, 2-3 (2005), 158–182.

[21] Yong Guo, Ana Lucia Varbanescu, Alexandru Iosup, Claudio Martella, and Theodore L Willke. 2014. Benchmarking graph-processing platforms: a vision. In *Proceedings of the 5th ACM/SPEC international conference on Performance engineering*. ACM, 289–292.

[22] Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen. 2014. Chronos: A Graph Engine for Temporal Graph Analysis. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)*. ACM, New York, NY, USA, Article 1, 14 pages. https://doi.org/10.1145/2592798.2592799

[23] Thomas Hartmann, Francois Fouquet, Matthieu Jimenez, Romain Rouvoy, and Yves Le Traon. 2017. Analyzing Complex Data in Motion at Scale with Temporal Graphs. In *The 29th International Conference on Software Engineering & Knowledge Engineering (SEKE'17)*. KSI Research, 6.

[24] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. 2010. The HiBench benchmark suite: Characterization of the MapReduce-based data analysis. In *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on*. IEEE, 41–51.

[25] IEEE. 2008. IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems. *IEEE Std 1588-2008 (Revision of IEEE Std 1588-2002)* (July 2008), 1–300. https://doi.org/10.1109/IEEESTD.2008.4579760

[26] Alexandru Iosup, Tim Hegeman, Wing Lung Ngai, Stijn Heldens, Arnau Prat-Pérez, Thomas Manhardto, Hassan Chafio, Mihai Capotă, Narayanan Sundaram, Michael Anderson, et al. 2016. Ldbc graphalytics: A benchmark for large-scale graph analysis on parallel and distributed platforms. *Proceedings of the VLDB Endowment* 9, 13 (2016), 1317–1328.

[27] 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.

[28] Raj Jain. 1991. *The Art of Computer Systems Performance Analysis - Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley.

[29] Ivo Jimenez, Michael Sevilla, Noah Watkins, Carlos Maltzahn, Jay Lofstead, Kathryn Mohror, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2017. The popper convention: Making reproducible systems evaluation practical. In *Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017 IEEE International*. IEEE, 1561–1570.

[30] Martin Junghanns, André Petermann, Martin Neumann, and Erhard Rahm. 2017. Management and Analysis of Big Graph Data: Current Systems and Open Challenges. In *Handbook of Big Data Technologies*. Springer, 457–505.

[31] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. 2005. Graphs over Time: Densification Laws, Shrinking Diameters and Possible Explanations. In *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining (KDD '05)*. ACM, New York, NY, USA, 177–187. https://doi.org/10.1145/1081870.1081893

[32] Min Li, Jian Tan, Yandong Wang, Li Zhang, and Valentina Salapura. 2015. Sparkbench: a comprehensive benchmarking suite for in memory data analytic platform spark. In *Proceedings of the 12th ACM International Conference on Computing Frontiers*. ACM, 53.

[33] Ruirui Lu, Gang Wu, Bin Xie, and Jingtong Hu. 2014. Stream bench: Towards benchmarking modern distributed stream computing frameworks. In *Utility and Cloud Computing (UCC), 2014 IEEE/ACM 7th International Conference on*. IEEE, 69–78.

[34] 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 *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD '10)*. ACM, New York, NY, USA, 135–146. https://doi.org/10.1145/1807167.1807184

[35] Andrew McGregor. 2014. Graph Stream Algorithms: A Survey. *SIGMOD Rec.* 43, 1 (May 2014), 9–20. https://doi.org/10.1145/2627692.2627694

[36] 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.

[37] Othon Michail and Paul G. Spirakis. 2018. Elements of the Theory of Dynamic Networks. *Commun. ACM* 61, 2 (Jan. 2018), 72–72. https://doi.org/10.1145/3156693

[38] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Manku, Chris Olston, Justin Rosenstein, and Rohit Varma. 2003. Query Processing, Resource Management, and Approximation in a Data Stream Management System–. In *IN CIDR*. Citeseer.

[39] Wing Lung Ngai, Tim Hegeman, Stijn Heldens, and Alexandru Iosup. 2017. Granula: Toward Fine-grained Performance Analysis of Large-scale Graph Processing Platforms. In *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems (GRADES'17)*. ACM, New York, NY, USA, Article 8, 6 pages. https://doi.org/10.1145/3078447.3078455

[40] Anil Pacaci, Alice Zhou, Jimmy Lin, and M. Tamer Özsu. 2017. Do We Need Specialized Graph Databases?: Benchmarking Real-Time Social Networking Applications. In *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems (GRADES'17)*. ACM, New York, NY, USA, Article 12, 7 pages. https://doi.org/10.1145/3078447.3078459

[41] Himchan Park and Min-Soo Kim. 2017. TrillionG: A Trillion-scale Synthetic Graph Generator Using a Recursive Vector Model. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. ACM, New York, NY, USA, 913–928. https://doi.org/10.1145/3035918.3064014

[42] Arnau Prat-Pérez, Joan Guisado-Gámez, Xavier Fernández Salas, Petr Koupy, Siegfried Depner, and Davide Basilio Bartolini. 2017. Towards a Property Graph Generator for Benchmarking. In *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems (GRADES'17)*. ACM, New York, NY, USA, Article 6, 6 pages. https://doi.org/10.1145/3078447.3078453

[43] Jari Saramäki and Esteban Moro. 2015. From seconds to months: an overview of multi-scale dynamics of mobile telephone calls. *The European Physical Journal B* 88, 6 (24 Jun 2015), 164. https://doi.org/10.1140/epjb/e2015-60106-6

[44] Yogesh Simmhan, Alok Kumbhare, Charith Wickramaarachchi, Soonil Nagarkar, Santosh Ravi, Cauligi Raghavendra, and Viktor Prasanna. 2014. GoFFish: A Sub-graph Centric Framework for Large-Scale Graph Analytics. In *Euro-Par 2014 Parallel Processing*, Fernando Silva, Inês Dutra, and Vítor Santos Costa (Eds.). Lecture Notes in Computer Science, Vol. 8632. Springer International Publishing, 451–462. https://doi.org/10.1007/978-3-319-09873-9_38

[45] Keval Vora, Rajiv Gupta, and Guoqing Xu. 2017. KickStarter: Fast and Accurate Computations on Streaming Graphs via Trimmed Approximations. *SIGARCH Comput. Archit. News* 45, 1 (April 2017), 237–251. https://doi.org/10.1145/3093337.3037748

[46] Lei Wang, Jianfeng Zhan, Chunjie Luo, Yuqing Zhu, Qiang Yang, Yongqiang He, Wanling Gao, Zhen Jia, Yingjie Shi, Shujie Zhang, et al. 2014. Bigdatabench: A big data benchmark suite from internet services. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*. IEEE, 488–499.

[47] Christo Wilson, Alessandra Sala, Krishna P. N. Puttaswamy, and Ben Y. Zhao. 2012. Beyond Social Graphs: User Interactions in Online Social Networks and Their Implications. *ACM Trans. Web* 6, 4, Article 17 (Nov. 2012), 31 pages. https://doi.org/10.1145/2382616.2382620

A ADDITIONAL MATERIAL

A.1 Graph Stream Generator Interface

Listing 1: User API of the Graph Stream Generator

```
bootstrapGlobalContext :: (): object
bootstrapGraph :: (graph: Graph, globalContext: object): void

nextEventType :: (globalContext: object): EventType
vertexSelect :: (eventType: EventType, globalContext: object):
      number
edgeSelect :: (eventType: EventType, globalContext: object):
      [number, number]

insertVertex :: (vertex: Vertex, globalContext: object): object
insertEdge :: (edge: Edge, globalContext: object): object
updateVertex :: (vertex: Vertex, globalContext: object): object
updateEdge :: (edge: Edge, globalContext: object): object
removeVertex :: (vertex: Vertex, globalContext: object): boolean
removeEdge :: (edge: Edge, globalContext: object): boolean

constraint :: (event: Event, globalContext: object): boolean
```

A.2 Description of the Experimental Evaluation Setups

Table 2: Configuration for Graph Stream Replayer test runs.

Configurations	
Setup	single instance (local only)
Machine	Intel Core i7-7700 (quad-core; 3.60 GHz), 32 GB of memory
Workload	generated social network workload
Targets	Pipe: `STDOUT` (stream replayer) to `STDIN` (measurement process) TCP: local socket to measurement process

Table 3: Configuration for the Weaver experiments.

Configurations	
Setup	two machines (1× replayer, 1× Weaver instance + local connector in Python using Weaver client API)
Machines	Intel Xeon E31220 CPU (quad-core; 3.10 GHz), 16 GB of memory, and 1 GigE NIC
Bootstrap Graph	BarabasiAlbert (n = 10000; m_0 = 250; M = 50)
Event Mix	`CREATE_VERTEX`: 10% `REMOVE_VERTEX`: 5% `UPDATE_VERTEX`: 35% `CREATE_EDGE`: 35% `REMOVE_EDGE`: 15% `UPDATE_EDGE`: 0%
Vertex Selection Functions	removing vertices: Zipf (based on degree; bias towards less connected vertices); updating vertices: uniform-random
Edge Selection Functions	source: uniform-random; target: Zipf (based on degree, bias towards strongly connected vertices)

Table 4: Configuration for the Chronograph experiments.

Configurations	
Setup	four machines (4× Chronograph shard workers; Kafka broker and stream replayer)
Machines	Intel Xeon E31220 CPU (quad-core; 3.10 GHz), 16 GB of memory, and 1 GigE NIC
Workload	converted LDBC SNB workload (only persons and connections); 190,518 events online influence rank computation
Stream	Base streaming rate: 2000 events/s, pause (20 sec) after 100,000 events, doubled rate between the 100,001th and 150,000th event