Practice of Streaming Processing of Dynamic Graphs: Concepts, Models, and Systems

Maciej Besta, Marc Fischer, Vasiliki Kalavri, Michael Kapralov, and Torsten Hoefler

Abstract—Graph processing has become an important part of various areas of computing, including machine learning, medical applications, social network analysis, computational sciences, and others. A growing amount of the associated graph processing workloads are *dynamic*, with millions of edges added or removed per second. Graph streaming frameworks are specifically crafted to enable the processing of such highly dynamic workloads. Recent years have seen the development of many such frameworks. However, they differ in their general architectures (with key details such as the support for the concurrent execution of graph updates and queries, or the incorporated graph data organization), the types of updates and workloads allowed, and many others. To facilitate the understanding of this growing field, we provide the first analysis and taxonomy of dynamic and streaming graph processing. We focus on identifying the fundamental system designs and on understanding their support for concurrency, and for different graph updates as well as analytics workloads. We also crystallize the meaning of different concepts associated with streaming graph processing, such as dynamic, temporal, online, and time-evolving graphs, edge-centric processing, models for the maintenance of updates, and graph databases. Moreover, we provide a bridge with the very rich landscape of graph streaming theory by giving a broad overview of recent theoretical related advances, and by discussing which graph streaming models and settings could be helpful in developing more powerful streaming frameworks and designs. We also outline graph streaming workloads and research challenges.

Index Terms—Streaming graphs, dynamic graphs, evolving graphs, streaming graph processing, dynamic graph processing, evolving graph processing, online graph processing, graph streaming frameworks, graph databases

1 Introduction

Analyzing massive graphs has become an important task. Example applications are investigating the Internet structure [39], analyzing social or neural relationships [20], or capturing the behavior of proteins [58]. Efficient processing of such graphs is challenging. First, these graphs are large, reaching even tens of trillions of edges [47], [102], [107], [108], [110], [130], [137], [158]. Second, the graphs in question are *dynamic*: new friendships appear, novel links are created, or protein interactions change. For example, 500 million new tweets in the Twitter social network appear per day, or billions of transactions in retail transaction graphs are generated every year [11].

Graph streaming frameworks such as GraphOne [103] or Aspen [56] emerged to enable processing and analyzing dynamically evolving graphs. Contrarily to static frameworks such as Ligra [79], [142], such systems execute graph analytics algorithms (e.g., PageRank) *concurrently* with graph updates (e.g., edge insertions). Thus, these frameworks must tackle unique challenges, for example effective modeling and storage of dynamic datasets, efficient ingestion of a stream of graph updates concurrently with graph queries, or support for effective programming model. In this work, we present the first taxonomy and analysis of such system aspects of the streaming processing of dynamic graphs.

We also crystallize the meaning of different concepts in streaming and dynamic graph processing. We investigate the notions of *temporal*, *time-evolving*, *online*, and *dynamic* graphs, as well as the differences between graph streaming frameworks and a related class of *graph database systems*.

We also analyze relations between the practice and the theory of streaming graph processing to facilitate incorporating recent theoretical advancements into the practical setting, to enable more powerful streaming frameworks. There exist different related theoretical settings, such as *streaming graphs* [115] or *dynamic graphs* [36] that come with different goals and techniques. Moreover, each of these settings comes with different *models*, for example the *dynamic graph stream* model [91] or the *semi-streaming* model [66]. These models assume different features of the processed streams, and they are used to develop provably efficient streaming algorithms. We analyze which theoretical settings and models are best suited for different practical scenarios, providing guidelines for architects and developers on what concepts could be useful for different classes of systems.

Next, we outline *models for the maintenance of updates*, such as the edge decay model [160]. These models are

- *Maciej Besta and Torsten Hoefler are with ETH Zurich, 8092 Zurich, Switzerland. E-mail: {maciej.besta, htor}@inf.ethz.ch.*
- *Marc Fischer is with PRODYNA (Schweiz) AG, 4052 Basel, Switzerland. E-mail: marc.fischer@prodyna.com.*
- *Vasiliki Kalavri is with Boston University, Boston, MA 02215 USA. E-mail: vkalavri@bu.edu.*
- *Michael Kapralov is with the School of Computer and Communication Sciences, EPFL, 1015 Lausanne, Switzerland. E-mail: michael.kapralov@epfl.ch.*

Manuscript received 31 December 2019; revised 30 October 2021; accepted 9 November 2021. Date of publication 30 November 2021; date of current version 8 May 2023.
This work was supported in part by the Google European Doctoral Fellowship and the European Research Council (ERC) under the European Union's Horizon 2020 programme grant agreement DAPP, under Grant 678880.
(Corresponding author: Maciej Besta.)
Recommended for acceptance by S. Pallickara.
Digital Object Identifier no. 10.1109/TPDS.2021.3131677

independent of the above-mentioned models for developing streaming algorithms. Specifically, they aim to define the way in which edge insertions and deletions are considered for updating different maintained structural graph properties such as distances between vertices. For example, the edge decay model captures the fact that edge updates from the past should *gradually* be made less relevant for the current status of a given structural graph property.

Finally, there are *general-purpose* dataflow systems such as Apache Flink [44] or Differential Dataflow [116]. We discuss the support for graph processing in such designs.

In general, we provide the following contributions:

- We crystallize the meaning of different concepts in dynamic and streaming graph processing, and we analyze the connections to the areas of graph databases and to the theory of streaming and dynamic graph algorithms.
- We provide the first taxonomy of graph streaming frameworks, identifying and analyzing key dimensions in their design, including data models and organization, concurrent execution, data distribution, targeted architecture, and others.
- We use our taxonomy to survey, categorize, and compare over graph streaming frameworks.
- We discuss in detail the design of selected frameworks.

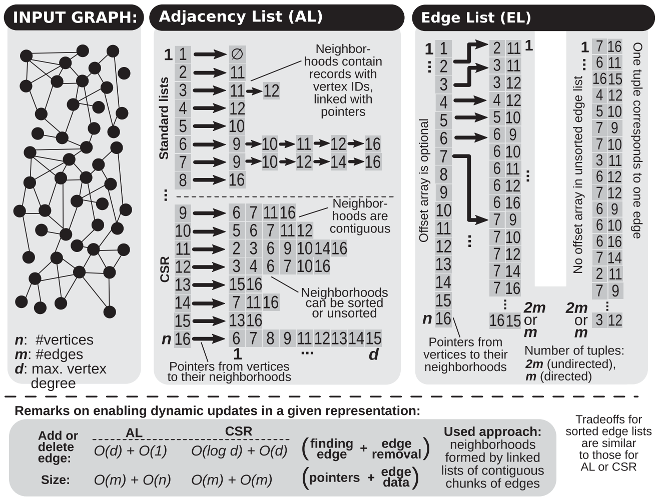
Complementary Surveys and Analyses. We provide the first taxonomy and survey on general streaming and dynamic graph processing. We complement related surveys on the *theory* of graph streaming models and algorithms [6], [115], [124], [164], analyses on *static* graph processing [18], [32], [59], [81], [114], [141], and on *general* streaming [90]. Finally, only one prior work summarized types of graph updates, partitioning of dynamic graphs, and some challenges [150].

2 Background and Notation

We first present concepts used in all the sections.

Graph Model. We model an undirected graph G as a tuple (V,E); $V=\{v_1,\ldots,v_n\}$ is a set of vertices and $E=\{e_1,\ldots,e_m\}\subseteq V\times V$ is a set of edges; $|V|=n$ and $|E|=m$. If G is directed, we use the name *arc* to refer to an edge with a direction. N_v denotes the set of vertices adjacent to vertex v, d_v is v's degree, and d is the maximum degree in G. If G is weighted, it is modeled by a tuple (V,E,w). Then, $w(e)$ is the weight of an edge $e\in E$. A weight is a single arbitrary number (e.g., an integer or a float).

Graph Representations. We also summarize fundamental *static* graph *representations*; they are used as a basis to develop dynamic graph representations in different frameworks. These are the *adjacency matrix* (AM), the *adjacency list* (AL), the *edge list* (EL), and the Compressed Sparse Row (CSR, sometimes referred to as Adjacency Array [41]).1 We illustrate these representations and we provide remarks on their dynamic variants in Fig. 1. In **AM**, a matrix $\mathbf{M}\in\{0,1\}^{n,n}$ determines the connectivity of vertices: $\mathbf{M}_{u,v}=1\Leftrightarrow$



INPUT GRAPH:
n: #vertices
m: #edges
d: max. vertex degree
Adjacency List (AL)
Standard lists
CSR
Neighbor-hoods contain records with vertex IDs, linked with pointers
Neighbor-hoods are contiguous
Neighborhoods can be sorted or unsorted
Pointers from vertices to their neighborhoods
Edge List (EL)
Offset array is optional
No offset array in unsorted edge list
One tuple corresponds to one edge
Pointers from vertices to their neighborhoods
Number of tuples: 2m (undirected), m (directed)
Remarks on enabling dynamic updates in a given representation:
Add or delete edge:
AL
CSR
O(d) + O(1)
O(log d) + O(d)
(finding edge + edge removal)
Size:
O(m) + O(n)
O(m) + O(m)
(pointers + edge data)
Used approach: neighborhoods formed by linked lists of contiguous chunks of edges
Tradeoffs for sorted edge lists are similar to those for AL or CSR

Fig. 1. **Illustration of fundamental graph representations**.

$(u,v)\in E$. In **AL**, each vertex u has an associated adjacency list A_u. This adjacency list maintains the IDs of all vertices adjacent to u. We have $v\in\mathrm{A}_u\Leftrightarrow(u,v)\in E$. AM uses $\mathcal{O}(n^2)$ space and can check connectivity of two vertices in $\mathcal{O}(1)$ time. AL requires $\mathcal{O}(n+m)$ space and it can check connectivity in $\mathcal{O}(|\mathrm{A}_u|)\subseteq\mathcal{O}(d)$ time. **EL** is similar to AL in the asymptotic time and space complexity as well as the general design. The main difference is that each edge is stored explicitly, with both its source and destination vertex. In AL and EL, a potential cause for inefficiency is scanning all edges to find neighbors of a given vertex. To alleviate this, index structures are employed [35]. Finally, **CSR** resembles AL but it consists of n *contiguous arrays* with neighborhoods of vertices. Each array is usually sorted by vertex IDs. CSR also contains a structure with offsets (or pointers) to each neighborhood array.

Graph Accesses. We often distinguish between *graph queries* and *graph updates*. A graph query (also called a *read*) may perform some computation on a graph and it returns information about the graph without modifying its structure. Such query can be *local*, also referred to as *fine* (e.g., accessing a single vertex or edge) or *global* (e.g., a PageRank analytics computation returning ranks of vertices). A graph update, also called a *mutation*, *modifies* the graph structure and/or attached labels or values (e.g., edge weights).

3 Clarification of Concepts and Areas

The term "graph streaming" has been used in different ways and has different meanings, depending on the context. We first extensively discuss and clarify these meanings, and we use this discussion to precisely illustrate the scope of our taxonomy and analyses. We illustrate all the considered concepts in Fig. 2. To foster developing more powerful and versatile systems for dynamic and streaming graph processing, we also summarize theoretical concepts.

3.1 Applied Dynamic and Streaming Graph Processing

We first outline the applied aspects and areas of dynamic and streaming graph processing.

1. Some works use CSR to describe a graph representation where all neighborhoods form a single contiguous array [103]. In this work, we use CSR to indicate a representation where each neighborhood is contiguous, but not necessarily all of them together.

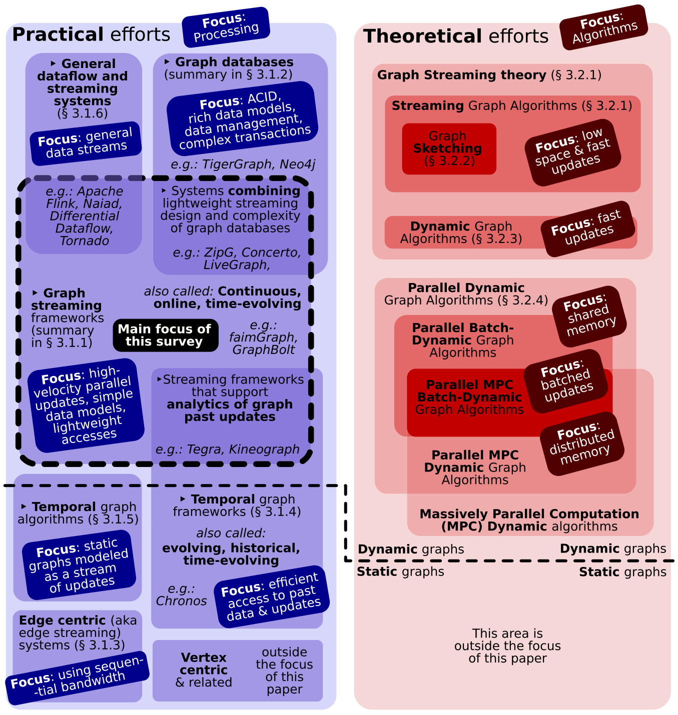

Practical efforts
Focus: Processing
General dataflow and streaming systems (§ 3.1.6)
Focus: general data streams
Graph databases (summary in § 3.1.2)
Focus: ACID, rich data models, data management, complex transactions
e.g.: TigerGraph, Neo4j
e.g.: Apache Flink, Naiad, Differential Dataflow, Tornado
Systems combining lightweight streaming design and complexity of graph databases
e.g.: ZipG, Concerto, LiveGraph,
Graph streaming frameworks (summary in § 3.1.1)
also called: Continuous, online, time-evolving
Main focus of this survey
e.g.: faimGraph, GraphBolt
Focus: high-velocity parallel updates, simple data models, lightweight accesses
Streaming frameworks that support analytics of graph past updates
e.g.: Tegra, Kineograph
Temporal graph algorithms (§ 3.1.5)
Focus: static graphs modeled as a stream of updates
Temporal graph frameworks (§ 3.1.4)
also called: evolving, historical, time-evolving
e.g.: Chronos
Focus: efficient access to past data & updates
Edge centric (aka edge streaming) systems (§ 3.1.3)
Focus: using sequen-tial bandwidth
Vertex centric & related
outside the focus of this paper
Theoretical efforts
Focus: Algorithms
Graph Streaming theory (§ 3.2.1)
Streaming Graph Algorithms (§ 3.2.1)
Graph Sketching (§ 3.2.2)
Focus: low space & fast updates
Dynamic Graph Algorithms (§ 3.2.3)
Focus: fast updates
Parallel Dynamic Graph Algorithms (§ 3.2.4)
Focus: shared memory
Parallel Batch-Dynamic Graph Algorithms
Focus: batched updates
Parallel MPC Batch-Dynamic Graph Algorithms
Focus: distributed memory
Parallel MPC Dynamic Graph Algorithms
Massively Parallel Computation (MPC) Dynamic algorithms
Dynamic graphs
Static graphs
Dynamic graphs
Static graphs
This area is outside the focus of this paper

Fig. 2. Overview of the *domains and concepts in the practice and theory of streaming and dynamic graph processing and algorithms.* This work focuses on *streaming graph processing* and its relations to other domains.

3.1.1 Streaming, Dynamic, and Time-Evolving Graphs

Many works [56], [63] use a term "streaming" or "streaming graphs" to refer to a setting in which a graph is *dynamic* [136] (also referred to as *time-evolving* [86], *continuous* [55], or *online* [69]) and it can be modified with updates such as edge insertions/deletions. *This setting is the primary focus of this survey*. In the work, we use "dynamic" to refer to the graph dataset being modified, and we reserve "streaming" to refer to the form of incoming graph accesses or updates. The time window of the associated queries in the online setting is of the form $[\text{Now} - \delta, \text{Now}]$ [87].

Closely related terms are *batch analytics* or *stream analytics*, used in relation to the computations and/or the *computation model* [10]. They refer to, respectively, running graph analytics *from scratch* (on static or dynamic data), and to running such analytics *incrementally*, on dynamic data. In this work, to comply with naming used in numerous works on dynamic graph processing, unless stated otherwise, we use the term "batch" to refer to *the ingestion of a certain number of graph updates together.*

3.1.2 Graph Databases and NoSQL Stores

Graph databases [31] are related to streaming and dynamic graph processing in that they support graph updates. Graph databases (both "native" graph database systems and NoSQL stores used as graph databases (e.g., RDF stores or document stores)) were described in detail in a recent work [31] and are beyond the main focus of this paper. However, there are numerous fundamental differences and similarities between graph databases and graph streaming frameworks, and we discuss these aspects in Section 6.

3.1.3 Streaming Processing of Static Graphs

Some works [34], [123], [131], [165] use "streaming" (also referred to as *edge-centric*) to indicate a setting in which the input graph is *static* but its edges are processed in a streaming fashion (as opposed to an approach based on random accesses into the graph data). Example associated frameworks are X-Stream [131], ShenTu [107], RStream [153], and several FPGA designs [34]. Such designs are outside the main focus of this survey; some of them were described by other works dedicated to static graph processing [34], [59].

3.1.4 Historical Graph Processing

There exist efforts into analyzing *temporal* (also referred to as historical or – somewhat confusingly – as *[time]-evolving*) graphs [82]. As noted by Dhulipala *et al.* [56], these efforts differ from streaming/dynamic/time-evolving graph analysis in that *one stores all past (historical) graph data to be able to query the graph as it appeared at any point in the past*. Contrarily, in streaming/dynamic/time-evolving graph processing, one focuses on keeping a graph in one (present) state. Additional snapshots are mainly dedicated to more efficient ingestion of graph updates, and *not* to preserving historical data for time-related analytics. Moreover, almost all works that focus solely on temporal graph analysis, for example the Chronos system [82], are *not* dynamic (i.e., they are *offline*): there is no notion of new incoming updates, but solely a series of past graph snapshots (instances). The time window of queries in historical graph processing is of the form $[T - \delta, T + \delta]$ [87], where T is some selected arbitrary point in the past. *These efforts are outside the focus of this survey* (we exclude these efforts, because they come with numerous challenges and design decisions (e.g., temporal graph models [163], temporal algebra [118], strategies for snapshot retrieval [159]) that require separate extensive treatment, while being unrelated to the streaming and dynamic graph processing). Still, *we describe concepts and systems that – while focusing on streaming processing of dynamic graphs, also enable keeping and processing historical data*. One such example is Tegra [87].

3.1.5 Temporal Graph Algorithms

Certain works analyze graphs where edges carry timing information, e.g., the order of communication between entities [156], [157]. One method to process such graphs is to *model them as a stream of incoming edges*, with the arrival time based on temporal information attached to edges. Thus, while being static graphs, their representation is dynamic. Thus, we picture these schemes as being partially in the dynamic setting in Fig. 2. These works come with no *frameworks*, and are outside the focus of our work.

3.1.6 General Dataflow and Streaming Systems

General streaming and dataflow systems, such as Apache Flink [44], Naiad [120], Tornado [140], or Differential Dataflow [116], can also be used to process dynamic graphs. However, most of the dimensions of our taxonomy are not well-defined for these general purpose systems. Overall, these systems provide a very general programming model and impose no restrictions on the format of streaming updates or graph state that the users construct. Thus, in principle, they could process queries and updates concurrently, support rich attached data, or even use transactional semantics. However, they do not come with pre-built features specifically targeting graphs.

3.2 Theory of Streaming and Dynamic Graphs

We next proceed to outline concepts in the theory of dynamic and streaming graph models and algorithms. Despite the fact that detailed descriptions are outside the scope of this paper, we firmly believe that explaining the associated general theoretical concepts and crystallizing their relations to the applied domain may facilitate developing more powerful streaming systems by – for example – incorporating efficient algorithms with provable bounds on their performance. In this section, we outline different theoretical areas and their focus. In general, in all the following theoretical settings, one is interested in maintaining (sometimes approximations to) a structural graph property of interest, such as connectivity structure, spectral structure, or shortest path distance metric, for graphs that are being modified by incoming updates (edge insertions and deletions).

3.2.1 Streaming Graph Algorithms

In *streaming graph algorithms* [49], [66], one usually starts with an empty graph with no edges (but with a fixed set of vertices). Then, at each algorithm step, a new edge is inserted into the graph, or an existing edge is deleted. Each such algorithm is parametrized by (1) *space complexity* (space used by a data structure that maintains a graph being updated), (2) *update time* (time to execute an update), (3) *query time* (time to compute an estimate of a given structural graph property), (4) *accuracy of the computed structural property*, and (5) *preprocessing time* (time to construct the initial graph data structure) [37]. Different streaming models can introduce additional assumptions, for example the Sliding Window Model provides restrictions on the *number of previous edges in the stream, considered for estimating the property* [49].

The goal is to develop algorithms that minimize different parameter values, with a special focus on *minimizing the storage for the graph data structure*. While space complexity is the main focus, significant effort is devoted to optimizing the runtime of streaming algorithms, specifically the time to process an edge update, as well as the time to recover the final solution (see, e.g., [105] and [95] for some recent developments). Typically the space requirement of graph streaming algorithms is $O(n \text{ polylog } n)$ (this is known as the semi-streaming model [66]), i.e., about the space needed to store a few spanning trees of the graph. Some recent works achieve "truly sublinear" space $o(n)$, which is sublinear in the number of vertices of the graph and is particularly good for sparse graphs [16], [17], [40], [65], [93], [94], [125]. The reader is referred to surveys on graph streaming algorithms [77], [115], [121] for more references.

3.2.2 Graph Sketching and Dynamic Graph Streams

Graph sketching [9] is an influential technique for processing graph streams with both insertions and deletions. The idea is to apply classical sketching techniques such as CountSketch [117] or distinct elements sketch (e.g., HyperLogLog [70]) to the edge incidence matrix of the input graph. Existing results show how to approximate the connectivity and cut structure [9], [13], spectral structure [95], [96], shortest path metric [9], [97], or subgraph counts [89], [91] using small sketches. Extensions to some of these techniques to hypergraphs were also proposed [78].

3.2.3 Dynamic Graph Algorithms

In the related area of *dynamic graph algorithms* one is interested in developing algorithms that approximate a combinatorial property of the input graph of interest (e.g., connectivity, shortest path distance, cuts, spectral properties) under edge insertions and deletions. Contrarily to graph streaming, in dynamic graph algorithms one puts less focus on minimizing space needed to store graph data. Instead, the primary goal is to *minimize time to conduct graph updates*. This has led to several very fast algorithms that provide updates with amortized poly-logarithmic update time complexity. See [19], [36], [45], [60], [62], [71], [149] and references within for some of the most recent developments.

3.2.4 Parallel Dynamic Graph Algorithms

Many algorithms were developed under the *parallel dynamic model*, in which a graph undergoes a series of incoming *parallel* updates. Next, the *parallel batch-dynamic model* is a recent development in the area of parallel dynamic graph algorithms [3], [4], [143], [147]. In this model, a graph is modified by updates coming *in batches*. A batch size is usually a function of n, for example $\log n$ or $\sqrt{n}$. Updates from each batch can be applied to a graph *in parallel*. The motivation for using batches is twofold: (1) incorporating parallelism into ingesting updates, and (2) reducing the cost per update. The associated algorithms focus on minimizing time to ingest updates into the graph while accurately maintaining a given structural graph property.

A *variant* [61] that *combines the parallel batch-dynamic model with the Massively Parallel Computation (MPC) model* [98] was also recently described. The MPC model is motivated by distributed frameworks such as MapReduce [53]. In this model, the maintained graph is stored on a certain number of machines (additionally assuming that the data in one batch fits into one machine). Each machine has a certain amount of space sublinear with respect to n. The main goal of MPC algorithms is to solve a given problem using $O(1)$ communication rounds while minimizing the volume of data communicated between the machines [98].

Finally, *another variant* of the MPC model that addresses dynamic graph algorithms *but without considering batches*, was also recently developed [84].

4 Taxonomy of Frameworks

We identify a taxonomy of graph streaming frameworks. We offer a detailed analysis of concrete frameworks using the taxonomy in Section 5 and in Tables 1 and 2. Overall, the identified taxonomy divides all the associated aspects into six classes: *ingesting updates (Section 4.2)*, *historical data maintenance (Section 4.3)*, *dynamic graph representation (Section 4.1)*, *incremental changes (Section 4.4)*, *programming API and models (Section 4.5)*, and *general architectural features*

TABLE 1
Comparison of Selected Representative Works

Reference	Ds?	Data location	Arch.	F?	Con?	B?	sB?	T?	acid?	P?	L?	S?	D?	Edge updates	Vertex updates	Remarks
STINGER [63]	✖	M-mem.	CPU	S	✖	■	■	✖	✖	◧	■	✖	✖	■ (A/R)	◧* (A/R)	*Removal is unclear
UNICORN [145]	■	M-mem.	CPU	C	✖	■	✖	✖	✖	✖	■	✖	✖	■ (A/R)	■ (A/R)	**Extends IBM InfoSphere Streams [38]**
DISTINGER [68]	■	M-mem.	CPU	S	✖	■	✖	✖	✖	✖	■	✖	✖	■ (A/R)	■ (A/R)	**Extends STINGER [63]**
cuSTINGER [77]	✖	GPU mem.	GPU*	S	✖	■	✖	✖	✖	✖	■	✖	✖	■ (A/R)	■ (A/R)	**Extends STINGER [63]**. *Single GPU.
EvoGraph [135]	✖	M-mem.	GPU*	C	✖	■	✖	✖	✖	◧	■	✖	✖	■ (A/R)	■ (A/R)	Supports multi-tenancy to share GPU resources. *Single GPU.
Hornet [42]	✖	GPU, M-mem.	GPU†	S	✖*	■	■	✖	✖	✖	■	✖	✖	■ (A/R/U)	■ (A/R/U)	*Not mentioned. †Single GPU
GraPU [140, 139]	■	M-mem., disk	CPU	C	✖	■	✖*	✖	✖	✖	■	✖	✖	■ (A/R)	✖	*Batches are processed with non-straightforward schemes
Grace [128]	✖	M-mem.	CPU	S+C	■ (s:C)	■	?	■	■	◧†	■	✖	✖	■ (A/R/U)	■ (A/R)	†To implement transactions
Kineograph [47]	■	M-mem.	CPU	C+S	■ (s:P)	■	✖	■	✖	■	■	■	■	■ (A/U*)	■ (A/U*)	*Custom update functions are possible
LLAMA [112]	✖	M-mem., disk	CPU	S	■ (s:C)	■	■	✖	✖	■	■	✖	✖	■ (A/R)	■ (A/R)	—
CellIQ [86]	■	Disk (HDFS)	CPU	C	■ (s)	?	✖	✖	✖	■	■	■	✖	■ (A/R)	■ (A/R)	**Extends GraphX [76] and Spark [162]**. *No details.
GraphTau [88]	■	M-mem., disk	CPU	C	■ (s)*	■	✖	✖	✖	■	■	■	✖	■ (A/R)	■ (A/R)	**Extends Spark**. *Offers **more** than simple snapshots.
DeltaGraph [55]	✖	M-mem.	CPU	C	■ (s:C)*	✖	✖	✖	✖	✖	■	✖	✖	■ (A/R)	■ (A/R)	*Relies on Haskell's features to create snapshots
GraphIn [136]	✖*	M-mem.	CPU	C+S	◧ (s)	■	✖	✖	✖	✖†	■	✖	✖	◧* (A/R)	◧* (A/R)	*Details are unclear. †Only mentioned
Aspen [56]	✖	M-mem., disk	CPU	S+C	■ (s:C)*	?	?	✖	✖	✖	■	✖	✖	■ (A/R)	■ (A/R)	*Focus on lightweight snapshots; enables **serializability**
Tegra [87]	■	M-mem., disk	CPU	C+S	■ (s)	?	?	✖	✖	■	◧*	■	?	■ (A/R)	■ (A/R)	**Extends Spark**. *Live updates are considered but outside core focus.
GraphInc [43]	■	M-mem., disk	CPU	C	■ (s)*	?	?	✖	✖	✖	■	✖	✖	■ (A/R/U)	■ (A/R/U)	**Extends Apache Giraph [14]**. *Keeps separate storage for the graph structure and for Pregel computations, but no details are provided.
ZipG [101]	■	M-mem.	CPU	S+C	■ (s)	?	?	✖	✖	◧	■	✖	✖	■ (A/R/U)	■ (A/R/U)	**Extends Spark & Succinct [4]**
GraphOne [104]	✖	M-mem.	CPU	S+C	■ (s:T)	■	■	✖	✖	◧	■	✖	✖	■ (A/R)	■ (A/R)	Updates of weights are possible
LiveGraph [166]	✖	M-mem., disk	CPU	S+C	■ (s:C)	✖	na	■	■	✖	■	✖	✖	■ (A/R/U)	■ (A/R/U)	—
Concerto [107]	■	M-mem.	CPU	S+C	■ (f)*	■	✖	■	■	✖	■	✖	✖	◧ (A/U)	◧ (A/U)	*A two-phase commit protocol based on fine-grained atomics
aimGraph [154]	✖	GPU mem.	GPU*	S+C	◧ (f)†	■	?	✖	✖	✖	■	✖	✖	■ (A/R)	✖	*Single GPU. †Only **fine** reads/updates are considered.
faimGraph [155]	✖	GPU, M-mem.	GPU*	S+C	◧ (f)†	■	■	✖	✖	✖	■	✖	✖	■ (A/R)	■ (A/R)	*Single GPU. †Only **fine** reads/updates, using locks/atomics.
GraphBolt [114]	✖	M-mem.	CPU	C+S	◧ (f)*	■	■	✖	✖	✖	■	✖	✖	■ (A/R)	■ (A/R)	**Uses Ligra [143]**. *Fine edge updates are supported.
DZiG [113]	✖	M-mem.	CPU	C+S	■ (f)	■	?	✖	✖	✖	■	✖	✖	■ (A/R)	■ (A/R)	
RisGraph [67]	✖	M-mem.	CPU	C/S	■ (sc)*	■†	?	✖	✖	■	■	✖	✖	■ (A/R)	◧ (A/R)	*Details in § 5.1.
GPMA (Sha [137])	◧*	GPU mem.	GPU*	S	◧ (o)†	■	?	✖	✖	✖	■	■	✖	■ (A/R)	✖	*Multiple GPUs within one server. †Details in § 5.1.
KickStarter [152]*	■	M-mem.	CPU	C	na*	■	na*	na*	na*	na*	■	na*	na*	■ (A/R)	?	**Uses ASPIRE [151]**. *It is a *runtime technique*.
Mondal et al. [120]	■	M-mem.*	CPU	C+S	◧†	?†	?†	■	■	✖	■	?†	?†	◧† (A)	◧† (A)	***Uses CouchDB as backend [11], †Unclear (relies on CouchDB)**
iGraph [89]	■	M-mem.	CPU	C	?	■	✖	✖	✖	✖	■	✖	✖	◧ (A/U)	◧ (A/U)	**Extends Spark**
Sprouter [1]	■	M-mem., disk	CPU	C	?	?	✖	✖	✖	✖	■	✖	✖	◧ (A)	?	**Extends Spark**

*They are grouped by the method of achieving concurrency between queries and updates (mutations). Within each group, the systems are sorted by publication date. "**Ds?**" (distributed): does a design target distributed environments such as clusters, supercomputers, or data centers? "**Data location**": the location of storing the processed dataset ("M-mem.": main memory; a system is primarily in-memory). "**Arch.**": targeted architecture. "**F**": focus on: computation (C), storage (S), computation and storage (C/S), mainly computation with some focus on storage (C+S), or mainly storage with some focus on computation (S+C). "**Con?**" (a method of achieving concurrent updates and queries): does a design support updates (e.g., edge insertions and removals) proceeding concurrently with queries that access the graph structure (e.g., edge lookups or PageRank computation). Whenever supported, we detail the method used for maintaining this concurrency: (s): snapshots (method unknown), (s:C): snapshots created with copy-on-write, (s:P): snapshots created periodically, (s:T): snapshots created with tombstones, (f): fine-grained synchronization, (sc): scheduling, (o): overlap. "**B?**" (batches): are updates batched? Batching entails support for data mutations at coarse granularity. "**sB?**" (sorted batches): can batches of updates be sorted for more performance? "**T?**" (transactions): are transactions supported? "**acid?**": are ACID transaction properties offered? "**P**": Does the system enable storing past graph snapshots? "**L?**" (live): are live updates supported (i.e., does a system maintain a graph snapshot that is "up-to-date": it continually ingests incoming updates)? "**S?**" (sliding): does a system support the Sliding Window Model for accessing past updates? "**D?**" (decay): does a system support the Decay Model for accessing past updates? "**Vertex / edge updates**": support for inserting and/or removing edges and/or vertices; "**A**": add, "**R**": remove, "**U**": update. "■": Support. "◧": Partial / limited support. "✖": No support. "?": Unknown.*

(Section 4.6). Due to space constraints, we focus on the *details of the system architecture* and we only sketch the *straightforward* taxonomy aspects (e.g., whether a system targets CPUs or GPUs) and list[2] them in Section 4.6.

4.1 Architecture of Dynamic Graph Representation

A core aspect of a streaming framework is the used representation of the maintained graph.

4.1.1 Used Fundamental Graph Representations

While the details of how each system maintains the graph dataset usually vary, the used representations can be grouped into a small set of fundamental types. Some frameworks use one of the *basic graph representations* (AL, EL, CSR, or AM) which are described in Section 2. Other graph representations are *based on trees*, where there is some additional hierarchical data structure imposed on the otherwise flat connectivity data; this hierarchical information is used to accelerate dynamic queries. Finally, frameworks constructed on top of more general infrastructure use a *representation provided by the underlying system*.

2. More details are in the extended paper version (see the link on page 1).

4.1.2 Blocking Within and Across Neighborhoods

In the taxonomy, we distinguish a common design choice in systems based on CSR or its variants. Specifically, one can combine the key design principles of AL and CSR by dividing each neighborhood into contiguous *blocks* (also referred to as *chunks*) that are larger than a single vertex ID (as in a basic AL) but smaller than a whole neighborhood (as in a basic CSR). This offers a tradeoff between flexible modifications in AL and more locality (and thus more efficient neighborhood traversals) in CSR [128]. Now, this blocking scheme is applied *within* each single neighborhood. We also distinguish a variant where multiple neighborhoods are grouped inside one block. We will refer to this scheme as blocking *across* neighborhoods. An additional optimization in the blocking scheme is to pre-allocate some reserved space at the end of each such contiguous block, to offer some number of fast edge insertions that do not require block reallocation. All these schemes are pictured in Fig. 3.

4.1.3 Supported Types of Vertex and Edge Data

Contrarily to graph databases that heavily use rich graph models such as the Labeled Property Graph [14], graph streaming frameworks usually offer simple data models, focusing on the *graph structure* and not on *rich data attached*

TABLE 2
Comparison of Selected Representative Works

Reference	Rich edge data	Rich vertex data	Tested analytics workloads	Fundamental representation	iB?	aB?	Id?	Ic?	PrM?	PrC?	Remarks
STINGER [63]	● (T, W, TS)	● (T)	CL, BC, BFS, CC, k-core	CSR	●	✗	● (a, d)	✗	● (sm)	✗	
Grace [128]	◐ (W)	✗	PR, CC, SSSP, BFS, DFS	CSR	◐*	✗	● (a)	✗	● (sm)	✗	* Due to partitioning of neighborhoods.
Concerto [107]	● (P)	● (P)	k-hop, k-core	CSR	?	●	●	✗	● (sm, tr*)	● (sa, i)*	* Graph views & event-driven processing
LLAMA [112]	● (P)	● (P)	PR, BFS, TC	CSR*	●	✗	● (a, t)	✗	● (sm)	✗	* multiversioned
DISTINGER [68]	● (T, W, TS)	● (T, W)	PR	CSR	●	✗	● (a, d)	✗	● (sm)	✗	—
cuSTINGER [77]	◐* (W, P, TS)	◐* (W, P)	TC	CSR	✗	✗	● (a, d)	✗	● (sm)	✗	* No details
aimGraph [154]	◐* (W)	◐* (W)	—	CSR*	●	✗	● (a)	✗	● (sm)	✗	* Resembles CSR.
Hornet [42]	◐ (W)	✗	BFS, SpMV, k-Truss	CSR	✗	●	● (a)	✗	● (sm)	✗	—
faimGraph [155]	● (W, P)	● (W, P)	PR, TC	CSR*	●	✗	● (a)	✗	● (sm)	✗	* Resembles CSR
LiveGraph [166]	● (T, P)	● (P)	PR, CC	CSR	● (g)	✗	● (a)	✗	● (sm)*	✗	* Primarily a data store
GraphBolt [114]	● (W)	✗	PR, BP, LP, CoEM, CF, TC	CSR	●	✗	● (a)	● (Rf/m,n)	● (sm)	● (sa*, i)	* Relies on BSP and Ligra's mappings
GraphIn [136]	✗	◐ (P)	BFS, CC, CL	CSR + EL	✗	✗	?	● (Rc/?)	● (sm)	● (sa, i)†	† Relies on GAS.
EvoGraph [135]	✗	✗	TC, CC, BFS	CSR + EL	✗	✗	?	● (Rc/?)	● (sm)	● (sa, i)	—
GraphOne [104]	◐ (W, T, P)	✗	BFS, PR, 1-Hop-query	CSR + EL	●	✗	● (a, t)	● (?/?)	● (sm, ss)	● (sa, p)	—
GraPU [140, 139]	◐ (W)	✗	BFS, SSSP, SSWP	AL*	✗	✗	● (a)	● (Rf†/m)	● (sm)	● (sa, i, sai)	* Relies on GoFS. † Relies on KickStarter
RisGraph [67]	● (W)	✗	CC, BFS, SSSP, SSWP	AL	?	?	● (a)	● (Rf*/m)	● (sm)	● (sa, p)	* Inspired by KickStarter
DZiG [113]	● (W)	✗	PR, BP, CoEM, CF, LP	AL	?	✗	● (a)	● (Rf/m,n)	● (sm)	● (sa, i)	
Kineograph [47]	✗	✗	TR, SSSP, k-exposure	KV store + AL*	?	✗	● (a)	● (Rc/?)	● (sm)	◐ (sa)†	* Details are unclear. † Uses vertex-centric
Mondal et al. [120]	✗	✗	—	KV store + documents*	✗	✗	● (a)	✗	● (sm)*	?*	* Relies on CouchDB
CellIQ [86]	● (P)	● (P)	Cellular specific	Collections (series)*	✗	✗	● (a, d)	● (Rc/?)	● (sm)	● (sa, i)†	* Uses RDDs. † Focus on geopartitioning
iGraph [89]	?	✗	PR	RDDs	✗	✗	●	● (Rc/?)	● (sm)	● (sa, i)*	* Relies on vertex-centric & BSP
GraphTau [88]	✗	✗	PR, CC	RDDs (series)	✗	✗	?	● (Rc/?) *	● (sm)	● (sa, i, p)†	† Relies on BSP & vertex-centric.
ZipG [101]	● (T, P, TS)	● (P)	TAO & LinkBench	Compressed flat files	✗	✗	● (a)	✗	● (sm)	✗	—
Sprouter [1]	?	?	PR	Tables*	✗	✗	?	✗	● (sm)	✗	* Relies on HGraphDB
DeltaGraph [55]	✗	✗	—	Inductive graphs*	✗	✗	✗	✗	● (sm, am)	● (sa)†	* Specific to functional languages [55]. † Mappings of vertices/edges
GPMA (Sha [137])	◐ (TS)	✗	PR, BFS, CC	Tree-based (PMA)	◐ (g)*	✗	● (a)	✗	● (sm)	✗	* A contiguous array with gaps in it
Aspen [56]	✗	✗*	BFS, BC, MIS, 2-hop, CL	Tree-based (C-Trees)	●	✗	● (a)	✗	● (sm)	◐ (sa)*	* Relies on Ligra
Tegra [87]	● (P)	● (P)	PR, CC	Tree-based (PART [52])	●*	✗	● (a)	● (Rc/m,n)	● (sm)	● (sa†, i, p)	* For properties. † Relies on GAS
GraphInc [43]	● (P)	● (P)	SSSP, CC, PR	?*	✗	✗	◐ (a)	● (Rc/?)	● (sm)	◐ (sa)*	* Uses Giraph's structures and model
UNICORN [145]	✗	✗	PR, RW	?*	✗	✗	?	●	● (sm)	● (sa, i)	* Uses InfoSphere
KickStarter [152]	◐ (W)	✗	SSWP, CC, SSSP, BFS	na*	na*	na*	na*	● (Rf/m)	● (sm)	na*	* Kickstarter is a *runtime technique*

*They are grouped by the used fundamental graph representation (within each group, by publication date). "**Rich edge/vertex data**": enabling additional data to be attached to an edge or a vertex ("T": type, "P": property, "W": weight, "TS": timestamp). "**Tested analytics workloads**": evaluated workloads beyond simple queries (**PR**: PageRank, **TR**: TunkRank, **CL**: clustering, **BC**: Betweenness Centrality, **CC**: Connected Components, **BFS**: Breadth-First Search, **SSSP**: Single Source Shortest Paths, **DFS**: Depth-First Search, **TC**: Triangle Counting, **SpMV**: Sparse matrix-vector multiplication, **BP**: Belief Propagation, **LP**: Label Propagation, **CoEM**: Co-Training Expectation Maximization, **CF**: Collaborative Filtering, **SSWP**: Single Source Widest Path, **TAO & LinkBench**: workloads used in Facebook's TAO and in LinkBench [15], **MIS**: Maximum Independent Set), **RW**: Random Walk. "**Fundamental Representation**": A key representation used to store the graph structure; all representation are explained in Section 4. "**iB**": Is blocking used to increase the locality of edges* within *the representation of a single neighborhood? "**(g)**": one uses empty* gaps *at the ends of blocks, to provide pre-allocated empty storage for faster edge insertions. "**aB**": Is blocking used to increase the locality of edges* across *different neighborhoods (i.e., can one store different neighborhoods within* one block*)? "**Id**": Is indexing used? "**(a)**": Indexing of the graph adjacency data, "**(d)**": Indexing of rich edge/vertex data, "**(t)**": Indexing of different graph snapshots, in the time dimension? "**Ic**": Are incremental changes supported? "**Rc**": incremental changes based on recomputation (the "offline approach"). "**Rf**": incremental changes based on refinement (the "online approach"). "**(m)**": Explicit support for* monotonic *algorithms in the context of incremental changes. "**(m,n)**": Explicit support for both* monotonic *and* non-monotonic *algorithms in the context of incremental changes. "**PrM**": Does the system offer a dedicated programming model (or API) related to graph* modifications*? "**(sm)**": API for simple graph modifications. "**(am)**": API for advanced graph modifications. "**(tr)**": API for triggered reactions to graph modifications. "**(ss)**": API for manipulating with the updates awaiting being ingested (e.g., stored in the log). "**PrC**": Does the system offer a dedicated programming model (or API) related to graph* computations*(i.e., analytics running on top of the graph being modified)? "**(sa)**": API for graph algorithms / analytics (e.g., PageRank) processing the main (i.e., up-to-date) graph snapshot. "**(p)**": API for graph algorithms / analytics (e.g., PageRank) processing the* past *graph snapshots. "**(i)**": API for incremental processing of graph algorithms / analytics. "**(sai)**" (i.e., (sa) + (i)): API for graph algorithms / analytics processing the* incremental changes themselves*. "●", "◐", "✗": A design offers a given feature, offers it in a limited way, and does not offer it, respectively. "?": Unknown.*

to vertices or edges. Still, different frameworks support basic additional vertex or edge data, most often *weights*. Next, in certain systems, both an edge and a vertex can have a *type* or an *attached property*. Finally, an edge can also have a *timestamp* that indicates the time of inserting this edge into the graph. A timestamp can also indicate a modification (e.g., an update of a weight of an existing edge). Details of such rich data are specific to each framework.

4.1.4 Other Indexing Structures

One uses indexing structures to accelerate different queries. In our taxonomy, we distinguish indices that speed up queries related to the *graph structure*, *rich data* (i.e., vertex or edge properties or labels), and *historic (temporal) aspects* (e.g., indices for edge timestamps).

4.2 Graph Storage Architecture and Mutations

The first core architectural aspect of any graph streaming framework are the details of its graph storage engines, and how incoming updates are ingested into it.

4.2.1 Concurrent Queries and Updates

We start with achieving concurrency between queries and updates (mutations).

One approach is based on *coarse-grained* synchronization (also referred to as *discretization*).

Here, one popular method is based on *snapshots*. Updates and queries are isolated from each other by making them execute on two different copies (snapshots) of the graph data. At some point, such snapshots are merged together. Depending on a system, the scope of data duplication (i.e., only a part of the graph may be copied into a new snapshot) and the details of merging may differ.

Snapshots can be created in different ways, for example with the well-known *copy-on-write* scheme, or *periodically* as determined by the underlying system details, or using *tombstones*.

In coarse-grained synchronization, one ingests updates, or resolves queries, *in batches*, i.e., multiple at a time, to amortize overheads from ensuring consistency of the maintained graph. We distinguish this design choice in the taxonomy because of its widespread use. Moreover, we identify a popular optimization in which a batch of edges to be removed or inserted is first *sorted* based on the ID of adjacent vertices. This introduces a certain overhead, but it also facilitates parallel ingestion of updates: updates associated with different vertices can be easier identified.

In *fine-grained* synchronization (also referred to as *continuous* updates), in contrast to coarse-grained synchronization (where updates are merged with the main graph representation during dedicated phases), updates are incorporated into the main dataset as soon as they arrive, often interleaved with queries, using synchronization protocols based on fine-grained locks and/or atomic operations. A variant of fine-grained synchronization is *Differential Dataflow* [116], where the ingestion strategy allows for concurrent updates and queries by relying on a combination of logical time, maintaining the knowledge of updates (referred to as deltas), and progress tracking. Specifically, the differential dataflow design operates on collections of key-value pairs enriched with timestamps and delta values. It views dynamic data as additions to or removals from input collections and tracks their evolution using logical time.

Finally, as also noted in past work [56], a system may simply not enable concurrency of queries and updates, and instead alternate between incorporating batches of graph updates and graph queries (i.e., updates are being applied to the graph structure while queries wait, and vice versa). This type of architecture may enable a high ratio of digesting updates as it does not have to resolve the problem of the consistency of graph queries running interleaved, concurrently, with updates being digested.

4.2.2 Transactional Support

We distinguish systems that support *transactions*, understood as units of work that enable isolation between concurrent accesses and correct recovery from potential failures. Moreover, some (but not all) systems ensure the *ACID semantics* of transactions.

4.3 Architecture of Historical Data Maintenance

While we do not focus on systems *solely* dedicated to the offline analysis of historical graph data, some streaming systems *enable different forms of accessing/analyzing such data*.

4.3.1 Storing Past Snapshots

In general, a streaming system may enable *storing past snapshots*, i.e., consistent past views (instances) of the whole dataset. Two general approaches for maintaining such past instances are (1) keeping snapshots themselves and (2) maintaining changes to the graph. The former approach makes deriving a given snapshot very efficient. However, it may come with storage overheads if many snapshots are maintained. The latter scheme reduces storage overheads, but it may be time-consuming because one has to reapply graph changes to construct a snapshot on demand.

4.3.2 Visibility of Past Graph Updates

There are several ways in which the information about past updates can be stored. Most systems only maintain a "live" version of the graph, where information about the past updates is not maintained,[3] in which all incoming graph updates are being incorporated into the structure of the maintained graph and they are all used to update or derive maintained structural graph properties. For example, if a user is interested in distances between vertices, then – in the snapshot model – the derived distances use *all* past graph updates. Formally, if we define the maintained graph at a given time t as $G_t = (V, E_t)$, then we have $E_t = \{e \mid e \in E \wedge t(e) \leq t\}$, where E are all graph edges and $t(e)$ is the timestamp of $e \in E$ [160].

Some streaming systems use the *sliding window model*, in which edges beyond certain moment in the past are being omitted when computing graph properties. Using the same notation as above, the maintained graph can be modeled as $G_{t,t'} = (V, E_{t,t'})$, where $E_{t,t'} = \{e \mid e \in E \wedge t \leq t(e) \leq t'\}$. Here, t and t' are moments in time that define the width of the *sliding window*, i.e., a span of time with graph updates that are being used for deriving certain query answers [160].

Both the snapshot model and the sliding window model do not reflect certain important aspects of the changing reality. The former takes into account all relationships *equally*, without distinguishing between the older and more recent ones. The latter enables omitting old relationships but does it *abruptly*, without considering the fact that certain connections may become *less relevant* in time but still *be present*. To alleviate these issues, the *edge decay model* was proposed [160]. In this model, each edge e (with a timestamp $t(e) \leq t$) has an independent probability $P^f(e)$ of being included in an analysis. $P^f(e) = f(t - t(e))$ is a non-decreasing *decay function* that determines *how fast edges age*. The authors of the edge decay model set f to be decreasing exponentially, with the resulting model being called the *probabilistic edge decay model*.

4.4 Architecture of Incremental Computation

A streaming framework may support an approach called "incremental changes" for faster convergence of graph algorithms. Assume that a certain graph algorithm is executed and produces some results, for example page ranks of each vertex. Now, the key observation behind the incremental changes is that the subsequent graph updates may not necessarily result in large changes to the derived page rank values. Thus, instead of recomputing the ranks from scratch, one can attempt to minimize the scope of recomputation, resulting in "incremental" changes to the ranking results. In our taxonomy, we will distinguish between supporting incremental changes in the *offline* (*"recomputation"*) or the *online* (*"refinement"*) mode. In the former, one updates analytics outcomes with *recomputation*. In the latter, one tracks dependencies (on-the-fly) between modified values and uses these dependencies to simply adjust the values that must be updated, from the point where the values become affected, without restarting computation. Recomputation based schemes may

3. This approach is sometimes referred to as the *"snapshot" model*. Here, the word "snapshot" means "a complete view of the graph, with all its updates". This naming is somewhat confusing, as "snapshot" can also mean "a specific copy of the graph generated for concurrent processing of updates and queries", cf. Section 4.2.

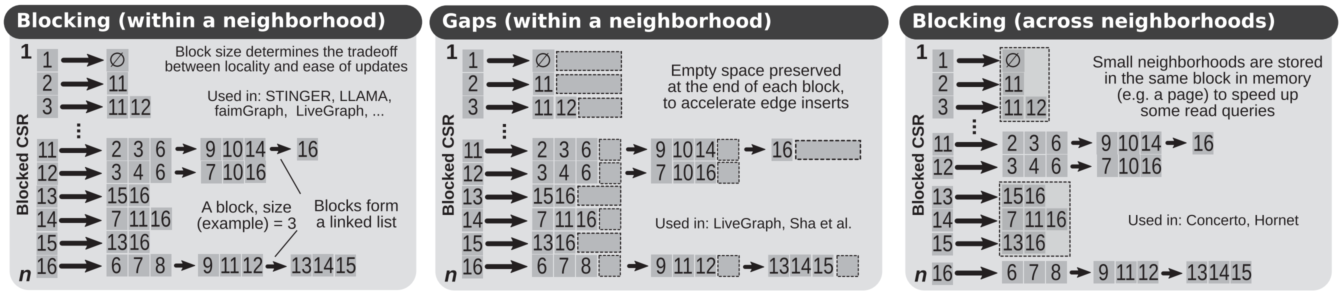

Blocking (within a neighborhood)
Block size determines the tradeoff between locality and ease of updates
Used in: STINGER, LLAMA, faimGraph, LiveGraph, ...
Blocked CSR
A block, size (example) = 3
Blocks form a linked list
Gaps (within a neighborhood)
Empty space preserved at the end of each block, to accelerate edge inserts
Used in: LiveGraph, Sha et al.
Blocking (across neighborhoods)
Small neighborhoods are stored in the same block in memory (e.g. a page) to speed up some read queries
Used in: Concerto, Hornet

Fig. 3. Illustration of blocking-related optimizatios in dynamic graph representations.

further differ in the amount of data that must be recomputed. For example, one may restart the computation from scratch for the whole graph upon mutations, or identify which vertices changed, and recompute precisely the values associated with these vertices.

4.5 Supported Programming APIs and Models

The final part of our taxonomy is the offered programming model and API. We identify two key classes of designs.

Graph Mutations. First, a framework may offer a selection of functions for *modifying the maintained graph*; such API may consist of *simple* basic functions (e.g., insert an edge) or *complex* ones (e.g., merge two graphs). Here, we additionally identify APIs for *triggered* events taking place upon specific updates, and for accessing and manipulating the *pending* graph updates (that await being ingested into the graph representation).

API for Graph Analytics. The second key API that a framework may support consists of functions for *running graph computations* on top of the maintained graph. Here, we identify specific APIs for controlling graph algorithms (e.g., PageRank) processing the *main (i.e., "live")* graph snapshot, or for controlling such computations running on top of *past* snapshots. Moreover, our taxonomy includes an API or models for *incremental* processing of the outcomes of graph algorithms (cf. Section 4.4).

4.6 General Architectural Features of Frameworks

The general features are the *location* of the maintained graph data (e.g., main memory or GPU memory), whether it is *distributed*, what is the *targeted hardware architecture* (general CPUs or GPUs), and whether a system is *general-purpose* or is it developed specifically for graph analytics.

5 Analysis of Frameworks

We now analyze existing frameworks using our taxonomy (cf. Section 4) in Tables 1 and 2, and in the following text.

We use symbols "▬", "▭", and "✗" to indicate that a given system offers a given feature, offers a given feature in a limited way, and does not offer a given feature, respectively.[4]

4. We encourage participation in this survey. In case the reader possesses additional information relevant for the tables, the authors would welcome the input. We also encourage the reader to send us any other information that they deem important, e.g., details of systems not mentioned in the current survey version.

5.1 Graph Storage Architecture and Mutations

We start with analyzing the method for achieving concurrency between updates and queries. Note that, with queries, we mean both local (fine) reads (e.g., fetching a weight of a given edge), but also global analytics (e.g., running PageRank) that also do not modify the graph structure.

First, most frameworks use snapshots. We observe that such frameworks have also some other snapshot-related design feature, for example Grace (uses snapshots also to implement transactions), GraphTau and Tegra (both support storing past snapshots), or DeltaGraph (harnesses Haskell's feature to create snapshots). The most popular mechanism for creating snapshots is copy-on-write, used in Grace, LLAMA, and others. The details (e.g., layouts or structures being copied) heavily depend on a specific system. Kineograph uses snapshots created periodically. GraphOne uses tombstones that mark which edges are to be removed, and thus could be swapped with new edges to be inserted. Second, a certain group of frameworks use fine-grained synchronization. The interleaving of updates and read queries is supported only with respect to fine reads (i.e., parallel *ingestion* of updates *while* running global analytics such as PageRank are not supported in the considered systems). Furthermore, two interesting methods for efficient concurrent ingestion of updates and queries have recently been proposed in the RisGraph system [68] and by Sha *et al.* [136]. The former uses *scheduling of updates*, i.e., the system uses fine-grained synchronization enhanced with a specialized scheduler that manipulates the ordering and timing of applying incoming updates to maximize throughput and minimize latency (different timings of applying updates may result in different performance penalties). In the latter, one *overlaps* the ingestion of updates with transferring the information about queries (e.g., over PCIe).

Most systems use batching, but only a few sort batches; the sorting overhead often exceeds benefits from faster ingestion. Next, only five frameworks use transactions, and four offer the ACID semantics of transactions. This illustrates that performance and high ingestion ratios are prioritized in the design of streaming frameworks over overall system robustness. Some frameworks that support ACID transactions rely with this respect on some underlying data store infrastructure: Sinfonia (for Concerto) and CouchDB (for the system by Mondal *et al.*). Others (Grace and LiveGraph) provide their own implementations of ACID.

5.2 Analysis of Support for Keeping Historical Data

Our analysis shows that reasonably many systems (>10) support keeping past data in some way. The details heavily depend on a given system. For example, Kineograph focuses on keeping past snapshots created periodically by the underlying runtime. Tegra enhances this approach by enabling the user to additionally create snapshots at arbitrary times.

To reduce both storage and performance overheads, the authors of Tegra observe that one could employ some combination of keeping snapshots and maintaining graph changes. Thus, performance would be improved as one would not have to start from scratch to arrive at a certain snapshot. Simultaneously, the memory pressure is reduced because not all snapshots are stored explicitly. However, this approach is not heavily explored in the literature. Systems such as STINGER or ZipG enable maintaining timestamps of graph mutations, which facilitates deriving the graph state at a selected point in time. However, these systems do not offer a dedicated API for such snapshot derivation, delegating such logic to the system user.

Systems keeping past snapshots often employ some additional form of reusing the graph structure across snapshots, to reduce memory overheads. For example, LLAMA employs a scheme in which parts of the graph, which are identical across the snapshots, are stored only once. Tegra uses a similar approach, with its Distributed Graph Snapshot Index. Some systems also use persistent storage to further alleviate the issue of maintaining multiple snapshots. An example such system is Tegra.

CelliQ, GraphTau, a system by Sha *et al.* , and Tegra also support the sliding window model. This is possible as they enable keeping past snapshots as well as obtaining the differences between these snapshots. Thus, the user can choose the range of past updates (e.g., incoming edges) when computing a given graph property. They also usually maintain indexing structures over historical data to accelerate fetching respective past instances. Tegra has a particularly rich set of features for analyzing historical data efficiently, approaching in its scope offline temporal frameworks such as Chronos [82]. Another system with a rich set of such features is Kineograph, the only one to support the exponential decay model of the visibility of past updates.

5.3 Analysis of Graph Representations

Most frameworks use some form of *CSR*. In certain cases, *CSR is combined with an EL* to form a dual representation; EL is often (but not exclusively) used in such cases as a log to store the incoming edges, for example in GraphOne. Certain other frameworks use **AL**, prioritizing the flexibility of graph updates over locality of accesses.

Most frameworks based on CSR use blocking within neighborhoods (i.e., each neighborhood consists of a linked list of contiguous blocks (chunks)). This enables a tradeoff between the locality of accesses and time to perform updates. The smaller the chunks are, the easier is to update a graph, but simultaneously traversing vertex neighborhoods requires more random memory accesses. Larger chunks improve locality of traversals, but require more time to update the graph structure. Two frameworks (Concerto and Hornet) use blocking across neighborhoods. This may help in achieving more locality whenever processing many small neighborhoods that fit in a block.

A few systems use *tree based* graph representations. For example, Sha *et al.* [136] use a variant of *packed memory array* (PMA), which is an array with all neighborhoods (i.e., essentially a CSR) augmented with an implicit binary tree structure for edge insertions and deletions in $O(\log^2 n)$ time.

Frameworks constructed on top of a more general infrastructure use a representation provided by the underlying system. For example, GraphTau [86], built on top of Apache Spark [162], uses the underlying abstraction called Resilient Distributed Datasets (RDDs) [161], [162]. RDDs can be implemented differently, for example using HDFS files [161]. Other frameworks use data representations that are harnessed by general processing systems or databases, for example KV stores, tables, or general collections.

All considered frameworks use some form of indexing. However, the indexes mostly keep the locations of vertex neighborhoods. Such an index is usually a simple array of size n, with cell i storing a pointer to the neighborhood N_i; this is a standard design for frameworks based on CSR. Whenever CSR is combined with blocking, a corresponding framework also offers the indexing of blocks used for storing neighborhoods contiguously. For example, this is the case for faimGraph and LiveGraph. Frameworks based on more complex underlying infrastructure benefit from indexing structures offered by the underlying system. For example, Concerto uses hash indexing offered by MySQL, and CellIQ and others can use structures offered by Spark. Finally, relatively few frameworks apply indexing of additional rich vertex or edge data, such as properties or labels. This is due to the fact that streaming frameworks, unlike graph databases, place more focus on the graph structure and much less on rich attached data. For example, STINGER indexes edges and vertices with given labels.

5.4 Analysis of Support for Incremental Changes

Around half of the considered frameworks support incremental changes to accelerate global graph analytics running on top of the maintained graph datasets. Frameworks that do not support them (e.g., faimGraph) usually put less focus on global analytics in the streaming setting.

Among systems that do support incremental computation, many are offline. These systems offer different mechanisms for detecting which vertices must be recomputed to update the analytics results to reflect recent graph mutations. This includes GraphIn, EvoGraph, Tegra, Kineograph, and others. Here, Tegra maintains is additionally able to incorporate incremental computation for *different* past snapshots, due to its focus on keeping and analyzing historical data.

Some systems are online, focusing on update refinement. For example, GraphBolt and KickStarter both carefully track dependencies between vertex values (that are being computed) and edge modifications. The differences between these two are driven by targeted classes of algorithms. GraphBolt assumes the Bulk Synchronous Parallel (BSP) [148] computation and thus ensures synchronous semantics. KickStarter instead focuses on path-based monotonic algorithms such as SSSP. It provides different optimizations. For example, it uses the fact that in many graph algorithms, the vertex value is simply *selected* from one single incoming edge. Unlike some other systems (e.g., Kineograph), GraphBolt and KickStarter enable performance gains also in the event of edge deletions, not only insertions. Finally, a very recent system called DZiG [112] improves the incremental capabilities of GraphBolt by utilizing the fact that in iterative graph algorithms, values of many vertices *stabilize* across iterations. This enables opportunities for annihilating unnecessary refinements. RisGraph applies KickStarter's approach for incremental computation to its design based on concurrent ingestion of fine-grained updates and queries.

Almost all the systems that support incremental changes focus on *monotonic* graph algorithms, i.e., algorithms, where the computed properties (e.g., vertex distances) are consistently either increasing or decreasing. Here, GraphBolt, DZiG, and Tegra also cover *non-monotonic* algorithms, such as Belief Propagation, Co-Training Expectation Maximization, or Collaborative Filtering.

5.5 Analysis of Offered Programming APIs and Models

Graph Mutations. We first analyze the supported APIs for *graph modifications*. All considered frameworks support a simple API for manipulating the graph, which includes operations such as adding or removing an edge. However, some frameworks offer more capabilities. We identify three such frameworks: Concerto, DeltaGraph, and GraphOne. Concerto has special functions for programming triggered events, i.e., events taking place automatically upon certain specific graph modifications. DeltaGraph offers functions for *merging* different graphs. Finally, GraphOne enables accessing and analyzing the updates that are still waiting (in a special log structure) to be ingested into the main graph structure. This can be used to apply some form of preprocessing of the updates, before they are applied to the main graph data, or to run some analytics on the updates.

Graph Analytics. We also discuss supported APIs for running global analytics on the maintained graph. First, we observe that a large fraction of frameworks do not support developing graph analytics at all. These systems, for example faimGraph, focus completely on graph mutations and local queries. However, other systems do offer an API for graph analytics (e.g., PageRank) processing the main (live) graph snapshot. These systems usually harness some existing programming model, for example Bulk Synchronous Parallel (BSP) [148]. Furthermore, frameworks that enable maintaining past snapshots, for example Tegra, also offer APIs for running analytics on such snapshots. These APIs are similar to the APIs for processing the main (live) graph versions, with a difference that the user also must identify the targeted specific past snapshot.

Finally, systems offering incremental changes also offer the associated APIs. Online systems such as GraphBolt and DZiG provide user-defined algorithm specific functions that enable refining aggregation values. Example functions are `propagate`, `retract`, or `repropagate`. The goal of these functions is to appropriately implement the logic of contributing to, or withdrawing from, vertex aggregation values. Offline systems often provide some way to indicate which vertices must be recomputed. For example, GraphIn and EvoGraph make the developer responsible for implementing a dedicated function that detects *inconsistent vertices*, i.e., vertices that became affected by graph updates. This function takes as arguments a batch of incoming updates and the vertex property related to the graph problem being solved (e.g., a parent in the BFS traversal problem). Whenever any update in the batch affects a specified property of some vertex, this vertex is marked as inconsistent, and is scheduled for recomputation. Another example is Tegra. It offers two functions, `diff` and `expand`. The former returns the difference (i.e., a modified subgraph) between two graph snapshots. The latter expands this subgraph with its 1–hop neighborhood. The resulting part of the graph is then scheduled for recomputation. A similar approach is used in other systems based on the underlying Spark infrastructure, i.e., CellIQ.

5.6 Supported Types of Graph Updates

Different systems support different forms of graph updates. The most widespread update is *edge insertion*, offered by all the considered systems. Second, *edge deletions* are supported by most frameworks. Finally, a system can also explicitly enable *adding* or *removing* a specified *vertex*. In the latter, a given vertex is removed with its adjacent edges.

5.7 Distributed Designs

Almost all the distributed frameworks rely on underlying existing backend infrastructure such as Spark (CelliQ, GraphTau, Tegra, ZipG, iGraph, Sprouter), CouchDB (work by Mondal *et al.*), or Giraph (GraphInc). Two frameworks that offer specialized distributed implementations are Kineograph and Concerto. Streaming frameworks rely on distribution mostly to enable scaling to larger datasets (by distributing a larger graph instance over multiple nodes) and to increase the throughput of graph queries (by distributing computation and update ingestion over multiple nodes). Furthermore, streaming frameworks rely on mature backends for effective fault tolerance.

5.8 Computation versus Storage

Some systems focus primarily on *computation* aspects of dynamic graph processing. For example, KickStarter offers an interesting model for incremental computation, while storage is outside its focus. Similarly, DZiG and GraphBolt focus on incremental computation, extending KickStarter's capabilities by – respectively – targeting BSP programs and by harnessing certain properties of such programs for more performance gains. Contrarily, systems such as Aspen focus on *storage*, usually by providing elaborate graph representations. Some systems, such as Tegra, come with enhancements into both aspects.

5.9 Analysis of Relations to Theoretical Models

First, despite the similarity of names, the (theoretical) field of *streaming graph algorithms* is *not* well connected to graph streaming frameworks: the focus of the former are fast algorithms operating with tight memory constraints that *by assumption* prevent from keeping the whole graph in memory, which is not often the case for the latter. Similarly, *graph sketching* focuses on approximate algorithms in a streaming setting, which is of little interest to streaming frameworks. On the other hand, the (theoretical) settings of *dynamic graph algorithms* and *parallel dynamic graph algorithms* are similar to that of the streaming frameworks. Their common assumption is that the whole maintained graph is available for queries (in-memory), which is also common for the streaming frameworks. Moreover, the *batch dynamic* model is even closer, as it explicitly assumes that edge updates arrive in batches, which reflects a common optimization in the streaming frameworks. We conclude that future developments in streaming frameworks could benefit from deepened understanding of the above mentioned theoretical areas. For example, one could use the recent parallel batch dynamic graph connectivity algorithm [3] in the implementation of any streaming framework, for more efficient connected components problem solution.

6 Graph Databases

Graph databases such as Neo4j [128] were introduced to alleviate performance overheads of querying graphs maintained as tables in relational databases; these overheads have been caused by the need to conduct many expensive joins when, for example, traversing a graph.

Streaming graph frameworks, similarly to graph databases, maintain a dynamically changing graph dataset under a series of updates and queries to the graph data. However, there are certain crucial differences that we now discuss. We refer the reader to a recent survey on the latter class of systems [31], which provides details of native graph databases such as Neo4j [128], RDF stores [48], and other types of NoSQL stores used for managing graphs.

6.1 Graph Databases versus Graph Streaming Systems

Targeted Workloads. Graph databases have traditionally focused on simple fine graph queries or updates, related to both the graph structure (e.g., verify if two vertices are connected) and the rich attached data (e.g., fetch the value of a given property) [64]. Another important class are "business intelligence" complex queries (e.g., fetch all vertices modeling cars, sorted by production year) [145]. Only recently, there has been interest in augmenting graph databases with capabilities to run global analytics such as PageRank [43]. In contrast, streaming frameworks focus on fine updates and queries, and on global analytics, but *not* on complex business intelligence queries. These frameworks put more focus on *high velocity updates* that can be rapidly ingested into the maintained. Next, of key interest are queries into the *structure* of the adjacency of vertices. This is often in contrast to graph databases, where many queries focus on the rich data attached to edges and vertices. These differences are reflected in all the following design aspects.

Ingesting Updates. Graph databases can use *many* different underlying designs (RDBMS style engines, native graph databases, KV stores, document stores, and others [31]), which means they may use different schemes for ingesting updates. However, a certain general difference between graph streaming frameworks and graph databases is that graph databases often include transactional support with ACID properties [31], [80], while very few streaming frameworks supports transactions and the ACID semantics of transactions. While most graph databases offer ACID, an example that does not is Cray Graph Engine [31]. The streaming graph updates, even if sometimes they also referred to as transactions [166], are usually "lightweight": single edge insertions or deletions, rather than arbitrary pattern matching queries common in graph database workloads. Overall, streaming frameworks focus on lightweight methods for fast and scalable ingestion of incoming updates, which includes optimizations such as batching of updates.

Graph Models and Representations. Graph databases usually deal with *complex and rich graph models* (such as the Labeled Property Graph [14] or Resource Description Framework [48]) where both vertices and edges may be of different types and may be associated with arbitrary rich properties such as pictures, strings, arrays of integers, or even data blobs. In contrast, models in streaming frameworks are usually simple, without support for arbitrary properties. This reflects the fact that the main focus in streaming frameworks is to investigate the structure of the maintained graph and its changes, and usually not rich attached data. This is also reflected by the associated indexing structures. While graph database systems maintain complex distributed index structures to accelerate different forms of queries over the rich attached data, streaming frameworks use simple index structures, most often only pointers to each vertex neighborhood, and very rarely additional structures pointing to edges/vertices with, e.g., common labels.

Data Distribution. Another interesting observation is support for *data replication* and *data sharding*. These two concepts refer to, respectively, the ability to replicate the maintained graph to more than one server (to accelerate certain read queries), and to partition the same single graph into several servers (to enable storing large graphs fully in-memory and to accelerate different types of accesses). Interestingly, streaming frameworks that enable distributed computation also support the more powerful but also more complex data sharding. Contrarily, while many distributed data stores used as graph databases (e.g., document stores) enable sharding as well, the class of "native" graph databases do not always support sharding. For example, the well-known Neo4j [128] graph databases only recently added support for sharding for *some* of its queries.

Keeping Historical Data. We observe that streaming frameworks often offer dedicated support for maintaining historical data, starting from simple forms such as dedicated edge insertion timestamps (e.g., in STINGER), to rich forms such as full historical data in a form of snapshots and different optimizations to minimize storage overheads (e.g., in Tegra). In contrast, graph databases most often do not offer such dedicated schemes. However, the generality of the used graph models facilitates maintaining such information at the user level (e.g., the user can use a timestamp label and/or property attached to each vertex or edge).

Incremental Changes. We do not know of any graph databases that offer explicit dedicated support for incremental changes. However, as most of such systems do not offer open source implementations, confirming this is hard. However, many streaming frameworks offer strong support for incremental changes, both in the form of its architecture and computational model tuned for this purpose, and its offered programming API. This is because incremental changes specifically target accelerating global graph analytics such as PageRank. These analytics have always been of key focus for streaming frameworks, and only recently became a relevant use case for graph databases [43].

Programming APIs and Models. Despite a lack of agreement on a single language for querying graph databases, all the languages (e.g., SPARQL [126], Gremlin [129], Cypher [72], [83], and SQL [50]) provide rich support for pattern matching queries [64] or business intelligence queries [145]. On the other hand, streaming frameworks do not offer such support. However, they do come with rich APIs for global graph analytics.

Summary. In summary, graph databases and streaming frameworks, despite different shared characteristics, are mostly complementary designs. Graph databases focus on rich data models and complex business intelligence workloads, while streaming frameworks' central interest are lightweight models and very fast update ingestion rates and global analytics. This can be seen in, for example, the design of the GraphTau framework, which explicitly offers an interface to load data for analytics *from a graph database*. Thus, using both systems together may often help to combine their advantages. Simultaneously, the gap between these two system classes is slowly shrinking, especially from the side of graph databases, where focus on global analytics and more performance can be seen in recent designs [43].

6.2 Systems Combining Both Areas

We describe example systems that provide features related to both graph streaming frameworks and graph databases.

Concerto [106] is a distributed in-memory graph store. The system presents features that can be found both in graph streaming frameworks (real-time graph queries and focus on fast, concurrent ingestion of updates) and in graph databases (triggers, ACID properties). It relies on Sinfonia [7], an infrastructure that provides a flat memory region over a set of distributed servers. Further, it offers ACID guarantees by distributed transactions (similar to the two-phase commit protocol) and writing logs to disk. The transactions are only short living for small operations such as reading and writing memory blocks; no transactions are available that consist of multiple updates. The graph data is stored by Sinfonia directly within in-memory objects that make up a data structure similar to an adjacency list. This data structure can also hold arbitrary properties.

ZipG [100] is a framework with focus on memory-efficient storage. It builds on Succint [5], a data store that supports random access to *compressed* unstructured data. ZipG exploits this feature and stores the graph in two files. The vertex file consists of the vertices that form the graph. Each row in the file contains the data related to one vertex, including the vertex properties. The edge file contains the edges stored in the graph. A single record in the edge file holds all edges of a particular type (e.g., a relationship or a comment in a social network) that are incident to a vertex. Further, this record contains all the properties of these edges. To enable fast access to the properties, metadata (e.g., lengths of different records, and offsets to the positions of different records) are also maintained by ZipG files. Succint compresses these files and creates immutable logs that are kept in main memory for fast access. Updates to the graph are stored in a single log store and compressed after a threshold is exceeded, allowing to run updates and queries concurrently. Pointers to the information on updates are managed such that logs do not have to be scanned during a query. Contrary to traditional graph databases, the system does not offer strict consistency or transactions.

Finally, *LiveGraph* [166] targets both transactional graph data management and graph analytics. Similarly to graph databases, it implements the property graph model and supports transactions, and similarly to analytics frameworks, it handles long running tasks that access the whole graph. For high performance, the system focuses on sequential data accesses. Vertices are stored in an array of vertex blocks on which updates are secured by a lock and applied using copy-on-write. For edges, a novel graph data structure is presented, called transactional edge log. Similar to an adjacency list there is a list of edges per vertex, but the data structure keeps all insertions, deletions and updates as edge log entries appended to the list. The data is stored in blocks, consisting of a header, edge log entries of fixed size and property entries (stored separately from the edge log entries). Each edge log entry stores the incident vertex, a create time and an update time. During a transaction, the reader receives a time stamp and reads only the data for which the create time is smaller than the given time stamp. Also the update time must be considered to omit stale data. Data is read starting from a tail pointer so a reader sees the updates first (no need to scan the old data). Further optimizations are applied, e.g., a Bloom filter allows to check quickly for existing edges. For an update, a writer must acquire a lock of the vertex. New data is appended on the tail of the edge log entries. Since the transaction edge log grows over time, a compression scheme is applied which is non-blocking for readers. The system guarantees persistence by writing data into a log and keeps changes locally until the commit phase, guaranteeing snapshot isolated transactions.

7 Performance Analysis

We now summarize key insights about performance of the described frameworks. We focus on (1) identifying the *fastest* frameworks, and on (2) understanding the performance effects of various design choices. Due to space constraints, we refer the reader to respective publications for the details of the evaluation setup. For concreteness, we report specific performance numbers, but the general performance patterns of the analyzed effects are similar for other input datasets and hardware architectures used in respective works. Our key source of data is a recent excellent broad analysis accompanying the evaluation of the DZiG processing system [112].

Summary of Performance-Oriented Goals. Two main performance goals of the studied frameworks are (1) maximizing the throughput of ingested updates, usually expressed in millions of inserted (or deleted) edges per second, and (2) accelerating graph analytics running on top of the maintained graph. Some systems (e.g., faimGraph [155]) only focus on maximizing the raw update rate. However, most systems attempt to maximizing the performance in both (1) *and* (2). Here, certain systems offer incremental changes (e.g., GraphBolt [113] or DZiG [112]) while others do not offer this capability, instead focusing on enhancing the schemes for incorporating graph mutations efficiently in the graph structures (e.g., Aspen [56] or GraphOne [103]).

Different results indicate that the former significantly outperform the latter when considering both (1) and (2) at the same time (i.e., in the *end-to-end runtime* comparisons of graph analytics such as PageRank or SSSP *and* simultaneous graph mutations) [112]. Here, we summarize the analysis in [112], which considers the following dimensions (the TwitterMPI graph): a targeted graph problem (PageRank with batched mutations, SSSP with batched mutations, and plain mutations), the size of graph mutation batches (1, 10, 100, 1k, 10k), and a framework (DZiG, GraphBolt, Aspen, GraphOne, LLAMA, STINGER). Now, in the DZiG analysis [112], for *plain mutations*, Aspen is the fastest on the above batch sizes (e.g., on a 32-core machine, Aspen achieves runtimes of 1e-4, 3e-4, 2e-3, 6e-3, 7e-3 for batches of 1, 10, 100, 1k, and 10k, respectively). When combining mutations and analytics, frameworks featuring dependency-driven incremental computation (GraphBolt, DZiG) outperform all other comparison targets, regardless of batch sizes and targeted problems. For example, for batch size 100, GraphBolt/DZiG use 11.7s/11.2s for PR and both take 0.06s for SSSP. Aspen takes 29.8s for PR and 3.31s for SSSP.

Systems such as Aspen come with more potential for the highest performance of *raw graph updates* [112]. These frameworks still try to minimize performance penalties when running graph analytics, compared to the running times of *static* graph processing frameworks. The highest performance of raw updates reported in the literature, without considering analytics, belongs to the GPU based faimGraph [155]. It achieves processing rates of nearly 200M edge updates / second (for batch size 1M, on several graphs such as coAuthorsD, on an NVIDIA Geforce GTX Titan Xp).

We also summarize performance patterns of techniques for *incremental computation*, using existing detailed analyses [112], [113]. First, dependency tracking (the online approach) systematically outperforms restarting computation upon graph mutations (the offline approach) [113]. Within the class of dependency tracking, differences between respective schemes depend on design details and targeted algorithms. For example, as expected, KickStarter outperforms GraphBolt on a non-BSP SSSP problem (consistent speedups of $\approx 7\times$ or more on the Twitter graph, for batch sizes of 1, 10, 100, 1k, 10k, on a single-socket 32-core machine), because GraphBolt, being tuned for BSP programs, ensures synchronous semantics, which is unnecessary for SSSP. Moreover, a very recent design indicates further opportunities for speedups within the class of dependency driven designs. Specifically, one can also utilize the fact that, in iterative graph algorithms, vertex values often *stabilize* after several iterations. This enables pruning unnecessary updates, and deliver speedups over other tuned dependency-driven systems that do not consider this effect [112]. We expect that this direction will be further explored in future works, for example by considering complex non-iterative and non-path-based graph mining workloads.

8 Challenges

Many research challenges related to streaming graph frameworks are similar to those in graph databases [31]. First, existing systems support numerous forms of data organization and types of graph representations, and it is unclear how to match these different schemes for different workload scenarios. A strongly related challenge, similarly to that in graph databases, is a high-performance system design for supporting both OLAP and OLTP style workloads. One can also try to accelerate different graph analytics problems in the streaming setting, for example graph coloring [21].

Second, while there is no consensus on a standard language for querying graph databases, even less is established for streaming frameworks. Different systems provide different APIs or programming abstractions [146]. Difficulties are intensified by a similar lack of consensus on most beneficial techniques for update ingestion and on computation models. This area is rapidly evolving and one should expect numerous new ideas, before a certain consensus is reached.

Moreover, contrarily to static graph processing, little research exists into accelerating streaming graph processing using hardware acceleration such as FPGAs [23], [34], [52], high-performance networking hardware and associated abstractions [24], [25], [28], [57], [73], [132], low-cost atomics [122], [133], hardware transactions [27], and others [8], [24]. One could also investigate topology-aware or routing-aware data distribution for graph streaming, especially together with recent high-performance network topologies [26], [101] and routing [22], [33], [74], [109]. Finally, ensuring speedups due to different data modeling abstractions, such as the algebraic abstraction [29], [30], [99], [104], may be a promising direction.

We also observe that, despite the fact that several streaming frameworks offer distributed execution and data sharding, the highest rate of ingestion is achieved by shared-memory single-node designs (cf. Section 7). An interesting challenge would be to make these designs distributed and to ensure that their ingestion rates increase even further, proportionally to the number of used compute nodes.

Finally, an interesting question is whether graph databases are inherently different from streaming frameworks. While merging these two classes of systems is an interesting ongoing effort, reflected by systems such as Graphflow [92] with many potential benefits, the difference in the associated workloads and industry requirements may be fundamentally different for a single unified solution.

9 Conclusion

Streaming and dynamic graph processing is an important research field. It is used to maintain numerous dynamic graph datasets, simultaneously ensuring high-performance graph updates, queries, and analytics workloads. Many graph streaming frameworks have been developed. They use different data representations, they are based on miscellaneous design choices for fast parallel ingestion of updates and resolution of queries, and they enable a plethora of queries and workloads. We present the first analysis and taxonomy of the rich landscape of streaming and dynamic graph processing. We crystallize a broad number of related concepts (both theoretical and practical), we list and categorize existing systems and discuss key design choices, we explain associated models, and we discuss related fields such as graph databases. Our work can be used by architects, developers, and project managers who want to select the most advantageous processing system or design, or simply understand this broad and fast-growing field.

Acknowledgments

The authors would like to thank PRODYNA AG (Darko Krizic, Jens Nixdorf, and Christoph Korner) for generous support, anonymous reviewers for useful comments, and Khuzaima Daudjee for suggestions on related work.

References

[1] Apache giraph project. Accessed: Nov. 18, 2021. [Online]. Available: https://giraph.apache.org/

[2] T. Abughofa and F. Zulkernine, "Sprouter: Dynamic graph processing over data streams at scale," in *Proc. 29th Int. Conf., DEXA*, 2018, pp. 321–328.

[3] U. A. Acar, D. Anderson, G. E. Blelloch, and L. Dhulipala, "Parallel batch-dynamic graph connectivity," in *Proc. 31st ACM Symp. Parallelism Algorithms Archit.*, 2019, pp. 381–392.

[4] U. A. Acar, A. Cotter, B. Hudson, and D. Türkoglu, "Parallelism in dynamic well-spaced point sets," in *ACM Symp. Parallelism Algorithms Archit.*, pp. 33–42, 2011.

[5] R. Agarwal *et al.*, "Succinct: Enabling queries on compressed data," in *Proc. Symp. Netw. Syst. Des. Implementation*, 2015, pp. 337–350.

[6] C. Aggarwal and K. Subbian, "Evolutionary network analysis: A survey," *ACM Comput. Surv.*, vol. 47, no. 1, 2014, Art. no. 10.

[7] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis, "Sinfonia: A new paradigm for building scalable distributed systems," in *ACM SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, pp. 159–174, 2007.

[8] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," *ACMSIGARCH Comp. Arch. News*, vol. 43, no. 3, pp. 105–117, 2016.

[9] K. J. Ahn, S. Guha, and A. McGregor, "Graph sketches: Sparsification, spanners, and subgraphs," in *Proc. ACM Symp. Princ. Database Syst.*, 2012, pp. 5–14.

[10] T. Akidau *et al.*, "The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing," *Proc. VLDB Endowment*, vol. 8, no. 12, pp. 1792–1803, 2015.

[11] K. Ammar, "Techniques and systems for large dynamic graphs," in *SIGMOD'16 PhD Symp.*, pp. 7–11, 2016.

[12] J. C. Anderson, J. Lehnardt, and N. Slater, *CouchDB: The Definitive Guide: Time to Relax*. Newton, MA, USA: O'Reilly Media, Inc., 2010.

[13] A. Andoni, J. Chen, R. Krauthgamer, B. Qin, D. P. Woodruff, and Q. Zhang, "On sketching quadratic forms," in *Proc. ACM Conf. Innov. Theor. Comput. Sci.*, 2016, pp. 311–319.

[14] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. Reutter, and D. Vrgoč, "Foundations of modern query languages for graph databases," *ACM Comput. Surv.*, vol. 50, no. 5, pp. 68:1–68:40, 2017.

[15] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan, "Linkbench: A database benchmark based on the facebook social graph," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2013, pp. 1185–1196.

[16] S. Assadi, S. Khanna, and Y. Li, "On estimating maximum matching size in graph streams," *Proc. Annu. ACM-SIAM Symp. Discrete Algorithms*, 2017.

[17] S. Assadi, S. Khanna, Y. Li, and G. Yaroslavtsev, "Maximum matchings in dynamic graph streams and the simultaneous communication model," in *Proc. 27th Annu. ACM-SIAM Symp. Discrete Algorithms*, 2016, pp. 1345–1364.

[18] O. Batarfi, R. El Shawi, A. G. Fayoumi, R. Nouri, A. Barnawi, S. Sakr *et al.*, "Large scale graph processing systems: Survey and an experimental evaluation," *Cluster Comput.*, vol. 18, no. 3, pp. 1189–1213, 2015.

[19] S. Behnezhad, M. Derakhshan, M. Hajiaghayi, C. Stein, and M. Sudan, "Fully dynamic maximal independent set with polylogarithmic update time," in *Proc. IEEE 60th Annu. Symp. Found. Comput. Sci.*, 2019, pp. 382–405.

[20] T. Ben-Nun, M. Besta, S. Huber, A. N. Ziogas, D. Peter, and T. Hoefler, "A modular benchmarking infrastructure for high-performance and reproducible deep learning," 2019, *arXiv:1901.10183*.

[21] M. Besta, A. Carigiet, K. Janda, Z. Vonarburg-Shmaria, L. Gianinazzi, and T. Hoefler, "High-performance parallel graph coloring with strong guarantees on work, depth, and quality," in *Proc. ACM/IEEE Supercomput.*, 2020, pp. 1–17.

[22] M. Besta *et al.*, "High-performance routing with multipathing and path diversity in ethernet and HPC networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 4, pp. 943–959, Apr. 2021.

[23] M. Besta, M. Fischer, T. Ben-Nun, J. De Fine Licht, and T. Hoefler, "Substream-centric maximum matchings on FPGA.," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, 2019, pp. 152–161.

[24] M. Besta, S. M. Hassan, S. Yalamanchili, R. Ausavarungnirun, O. Mutlu, and T. Hoefler, "Slim noc: A low-diameter on-chip network topology for high energy efficiency and scalability," *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 43–55, 2018.

[25] M. Besta and T. Hoefler, "Fault tolerance for remote memory access programming models," in *Proc. ACM 23rd Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2014, pp. 37–48.

[26] M. Besta and T. Hoefler, "Slim fly: A cost effective low-diameter network topology," in *Proc. ACM/IEEE Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2014, pp. 348–359.

[27] M. Besta and T. Hoefler, "Accelerating irregular computations with hardware transactional memory and active messages," in *Proc. 24th Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2015, pp. 161–172.

[28] M. Besta and T. Hoefler, "Active access: A mechanism for high-performance distributed data-centric computations," in *Proc. ACM Int. Conf. Supercomput.*, 2015, pp. 155–164.

[29] M. Besta *et al.*, "Communication-efficient jaccard similarity for high-performance distributed genome comparisons," 2019, *arXiv:1911.04200*.

[30] M. Besta, F. Marending, E. Solomonik, and T. Hoefler, "Slimsell: A vectorizable graph representation for breadth-first search," in *IEEE IPDPS*, pp. 32–41, 2017.

[31] M. Besta *et al.*, "Demystifying graph databases: Analysis and taxonomy of data organization, system designs, and graph queries," 2019, *arXiv:1910.09017*.

[32] M. Besta, M. Podstawski, L. Groner, E. Solomonik, and T. Hoefler, "To push or to pull: On reducing communication and synchronization in graph computations," in *Proc. 24th Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2017, pp. 93–104.

[33] M. Besta *et al.*, "FatPaths: Routing in supercomputers and data centers when shortest paths fall short," in *Proc. 24th Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2019, pp. 1–18.

[34] M. Besta, D. Stanojevic, J. D. F. Licht, T. Ben-Nun, and T. Hoefler, "Graph processing on FPGAs: Taxonomy, survey, challenges," 2019, *arXiv:1903.06697*.

[35] M. Besta, D. Stanojevic, T. Zivic, J. Singh, M. Hoerold, and T. Hoefler, "Log (graph): A near-optimal high-performance graph representation," in *Proc. 27th Int. Conf. Parallel Archit. Compilation Techn.*, 2018, Art. no. 7.

[36] S. Bhattacharya, M. Henzinger, and D. Nanongkai, "A new deterministic algorithm for dynamic set cover," in *Proc. IEEE Annu. Symp. Found. Comput. Sci.*, 2019, pp. 406–423.

[37] S. Bhattacharya, M. Henzinger, D. Nanongkai, and C. Tsourakakis, "Space-and time-efficient algorithm for maintaining dense subgraphs on one-pass dynamic streams," in *Proc. 47th Annu. ACM Symp. Theory Comput.*, 2015, pp. 173–182.

[38] A. Biem *et al.*, "IBM infosphere streams for scalable, real-time, intelligent transportation services," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2010, pp. 1093–1104.

[39] P. Boldi and S. Vigna, "The webgraph framework i: Compression techniques," in *Proc. ACM World Wide Web*, 2004, pp. 595–602.
[40] M. Bury *et al.*, "Structural results on matching estimation with applications to streaming," *Algorithmica*, vol. 81, no. 1, pp. 367–392, 2019.
[41] F. Busato, O. Green, N. Bombieri, and D. A. Bader, "HorNet: An efficient data structure for dynamic sparse graphs and matrices on GPUs," in *Proc. IEEE Conf. High Perform. Extreme Comput.*, 2018, pp. 1–7.
[42] Z. Cai, D. Logothetis, and G. Siganos, "Facilitating real-time graph mining," in *Proc. ACM CloudDB*, 2012, pp. 1–8.
[43] M. Capotă, T. Hegeman, A. Iosup, A. Prat-P érez, O. Erling, and P. Boncz, "Graphalytics: A big data benchmark for graph-processing platforms," in *Proc. GRADES'15*, pp. 1–6, 2015.
[44] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *IEEE-Comput. Soc. Bull. Tech. Committee Data Eng.*, vol. 38, no. 4, pp. 28–38, 2015.
[45] S. Chechik and T. Zhang, "Fully dynamic maximal independent set in expected poly-log update time," in *Proc. IEEE Annu. Symp. Found. Comput. Sci.*, 2019, pp. 370–381.
[46] R. Cheng *et al.*, "Kineograph: Taking the pulse of a fast-changing and connected world," in *Proc. ACM EuroSys*, 2012, pp. 85–98.
[47] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, "One trillion edges: Graph processing at facebook-scale," in *Proc. VLDB Endowment*, vol. 8, no. 12, pp. 1804–1815, 2015.
[48] R. Cyganiak, D. Wood, and M. Lanthaler, "RDF 1.1 Concepts and Abstract Syntax," Accessed: Jul. 5, 2018. [Online]. Available: https://www.w3.org/TR/rdf11-concepts/
[49] M. Datar, A. Gionis, P. Indyk, and R. Motwani, "Maintaining stream statistics over sliding windows," *SIAM J. Comput.*, vol. 31, no. 6, pp. 1794–1813, 2002.
[50] C. J. Date and H. Darwen, *A Guide to the SQL Standard*. vol. 3, New York, NY, USA: Addison-Wesley, 1987.
[51] A. Dave, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "Persistent adaptive radix trees: Efficient fine-grained updates to immutable data."
[52] J. de Fine Licht *et al.*, "Transformations of high-level synthesis codes for high-performance computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 5, pp. 1014–1029, 2020.
[53] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.
[54] P. Dexter, Y. D. Liu, and K. Chiu, "Lazy graph processing in Haskell," *ACM SIGPLAN Notices*, vol. 51, pp. 182–192, 2016.
[55] P. Dexter, Y. D. Liu, and K. Chiu, "Formal foundations of continuous graph processing," 2019, *arXiv:1911.10982*.
[56] L. Dhulipala *et al.*, "Low-latency graph streaming using compressed purely-functional trees," 2019, *arXiv:1904.08380*.
[57] S. Di Girolamo *et al.*, "Network-accelerated non-contiguous memory transfers," 2019, *arXiv:1908.08590*.
[58] L. Di Paola, M. De Ruvo, P. Paci, D. Santoni, and A. Giuliani, "Protein contact networks: An emerging paradigm in chemistry," *Chem. Rev.*, vol. 113, no. 3, pp. 1598–1613, 2012.
[59] N. Doekemeijer and A. L. Varbanescu, "A survey of parallel graph processing frameworks," Delft Univ. Technol., Delft, The Netherlands, Tech. Rep. PDS-2014–003, 2014.
[60] R. Duan, H. He, and T. Zhang, "Dynamic edge coloring with improved approximation," in *Proc. Annu. ACM-SIAM Symp. Discrete Algorithms*, 2019, pp. 1937–1945.
[61] D. Durfee, L. Dhulipala, J. Kulkarni, R. Peng, S. Sawlani, and X. Sun, "Parallel batch-dynamic graphs: Algorithms and lower bounds," *Proc. Annu. ACM-SIAM Symp. Discrete Algorithms*, 2020.
[62] D. Durfee, Y. Gao, G. Goranci, and R. Peng, "Fully dynamic spectral vertex sparsifiers and applications," in *Proc. Annu. ACM Symp. Theory Comput.*, 2019, pp. 914–925.
[63] D. Ediger, R. McColl, J. Riedy, and D. A. Bader, "Stinger: High performance data structure for streaming graphs," in *Proc. IEEE Conf. High Perform. Extreme Comput.*, 2012, pp. 1–5.
[64] O. Erling *et al.*, "The LDBC Social network benchmark: Interactive workload," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2015, pp. 619–630.
[65] H. Esfandiari, M. Hajiaghayi, V. Liaghat, M. Monemizadeh, and K. Onak, "Streaming algorithms for estimating the matching size in planar graphs and beyond," *ACM Trans. Algorithms*, vol. 14, no. 4, pp. 1–23, 2018.
[66] J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang, "On graph problems in a semi-streaming model," *Theor. Comput. Sci.*, vol. 348, no. 2–3, pp. 207–216, 2005.
[67] G. Feng *et al.*, "DISTINGER: A distributed graph data structure for massive dynamic graph processing," in *Proc. IEEE Int. Conf. Big Data*, 2015, pp. 1814–1822.
[68] G. Feng *et al.*, "Risgraph: A real-time streaming system for evolving graphs to support sub-millisecond per-update analysis at millions ops/s," in *Proc. Int. Conf. Manage. Data*, 2021, pp. 513–527.
[69] I. Filippidou and Y. Kotidis, "Online and on-demand partitioning of streaming graphs," in *Proc. IEEE Int. Conf. Big Data*, 2015, pp. 4–13.
[70] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier, "HYPERLOGLOG: The analysis of a near-optimal cardinality estimation algorithm," in *Proc. Discrete Math, Theor. Comput. Sci.*, 2007, pp. 137–156.
[71] S. Forster and G. Goranci, "Dynamic low-stretch trees via dynamic low-diameter decompositions," in *Proc. Annu. ACM Symp. Theory Comput.*, 2019, pp. 377–388.
[72] N. Francis *et al.*, "Cypher: An evolving query language for property graphs," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2018, pp. 1433–1445.
[73] R. Gerstenberger *et al.*, "Enabling highly-scalable remote memory access programming with MPI-3 one sided," in *Proc. ACM/IEEE Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2013, p. 1.
[74] S. Ghorbani, Z. Yang, P. Godfrey, Y. Ganjali, and A. Firoozshahian, "Drill: Micro load balancing for low-latency data center networks," in *Proc. Annu. Conf. ACM Special Interest Group Data Commun.*, 2017, pp. 225–238.
[75] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework," in *Proc. 11th USENIX Conf. Operating Syst. Des. Implementation*, 2014, pp. 599–613.
[76] O. Green and D. A. Bader, "cuSTINGER: Supporting dynamic graph algorithms for GPUs," in *Proc. IEEE IEEE Conf. High Perform. Extreme Comput.*, 2016, pp. 1–6.
[77] S. Guha and A. McGregor, "Graph synopses, sketches, and streams: A survey," *Proc. VLDB Endowment*, vol. 5, no. 12, pp. 2030–2031, 2012.
[78] S. Guha, A. McGregor, and D. Tench, "Vertex and hyperedge connectivity in dynamic graph streams," in *ACM Symp. Princ. Database Syst.*, 2015, pp. 241–247.
[79] M. Han and K. Daudjee, "Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing systems," in *Proc. VLDB Endowment*, vol. 8, no. 9, pp. 950–961, 2015.
[80] M. Han and K. Daudjee, "Providing serializability for pregel-like graph processing systems," in *Proc. Annu. Int. Conf. Extending Database Technol.*, 2016, pp. 77–88.
[81] M. Han, K. Daudjee, K. Ammar, M. T. Özsu, X. Wang, and T. Jin, "An experimental comparison of pregel-like graph processing systems," in *Proc. VLDB Endowment*, vol. 7, no. 12, pp. 1047–1058, 2014.
[82] W. Han *et al.*, "Chronos: A graph engine for temporal graph analysis," in *Proc. 9th Eur. Conf. Comput. Syst.*, 2014, pp. 1–14.
[83] F. Holzschuher and R. Peinl, "Performance of graph query languages: Comparison of cypher, gremlin and native access in neo4j," in *Proc. ACM Joint EDBT/ICDT Workshops*, 2013, pp. 195–204.
[84] G. F. Italiano, S. Lattanzi, V. S. Mirrokni, and N. Parotsidis, "Dynamic algorithms for the massively parallel computation model," in *Proc. ACM ACM Symp. Parallelism Algorithms Archit.*, 2019, pp. 49–58.
[85] A. Iyer, L. E. Li, and I. Stoica, "CelliQ: Real-time cellular network analytics at scale," in *Proc. Symp. Netw. Syst. Des. Implementation*, 2015, pp. 309–322.
[86] A. P. Iyer, L. E. Li, T. Das, and I. Stoica, "Time-evolving graph processing at scale," in *Proc. 4th Int. Workshop Graph Data Manage. Experiences Syst.*, 2016, pp. 1–6.
[87] A. P. Iyer, Q. Pu, K. Patel, J. E. Gonzalez, and I. Stoica, "TEGRA: Efficient ad-hoc analytics on evolving graphs," in *Proc. Symp. Netw. Syst. Des. Implementation*, 2021, pp. 337–355.
[88] W. Ju, J. Li, W. Yu, and R. Zhang, "iGraph: An incremental data processing system for dynamic graph," *Front. Comput. Sci.*, vol. 10, no. 3, pp. 462–476, 2016.

[89] J. Kallaugher, M. Kapralov, and E. Price, "The sketching complexity of graph and hypergraph counting," *IEEE Annu. Symp. Found. Comput. Sci.*, 2018, pp. 556–567.

[90] S. Kamburugamuve and G. Fox, "Survey of distributed stream processing," Bloomington: Indiana University, Bloomington, IN, USA, 2016.

[91] D. M. Kane, K. Mehlhorn, T. Sauerwald, and H. Sun, "Counting arbitrary subgraphs in data streams," in *Proc. Int. Colloquium Automata, Lang., Program.*, 2012, pp. 598–609.

[92] C. Kankanamge, S. Sahu, A. Mhedbhi, J. Chen, and S. Salihoglu, "Graphflow: An active graph database," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2017, pp. 1695–1698.

[93] M. Kapralov, S. Khanna, and M. Sudan, "Approximating matching size from random streams," in *Proc. Annu. ACM-SIAM Symp. Discrete Algorithms*, 2014.

[94] M. Kapralov, S. Mitrovic, A. Norouzi-Fard, and J. Tardos, "Space efficient approximation to maximum matching size from uniform edge samples, in *Proc. Annu. ACM-SIAM Symp. Discrete Algorithms*, 2020.

[95] M. Kapralov, A. Mousavifar, C. Musco, C. Musco, N. Nouri, A. Sidford, and J. Tardos, "Fast and space efficient spectral sparsification in dynamic streams," in *Proc. Annu. ACM-SIAM Symp. Discrete Algorithms*, 2020.

[96] M. Kapralov, N. Nouri, A. Sidford, and J. Tardos, "Dynamic streaming spectral sparsification in nearly linear time and space," 2019, *arXiv:1903.12150v1*.

[97] M. Kapralov and D. P. Woodruff, "Spanners and sparsifiers in dynamic streams," in *Proc. ACM Symp. Princ. Distrib. Comput.*, 2014, pp. 272–281.

[98] H. Karloff, S. Suri, and S. Vassilvitskii, "A model of computation for MapReduce," in *Proc. Annual ACM-SIAM Symp. Discrete Algorithms*, 2010, pp. 938–948.

[99] J. Kepner, P. Aaltonen, D. Bader, A. Buluç, F. Franchetti, J. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke, *et al.*, "Mathematical foundations of the graphblas," in *Proc. IEEE High Perform. Extreme Comput. Conf.*, 2016, pp. 1–9.

[100] A. Khandelwal, Z. Yang, E. Ye, R. Agarwal, and I. Stoica, "ZipG: A memory-efficient graph store for interactive queries," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2017, pp. 1149–1164.

[101] J. Kim, W. J. Dally, S. Scott, and D. Abts, "Technology-driven, highly-scalable dragonfly topology," in *Proc. Int. Symp. Comput. Archit.*, 2008, pp. 77–88.

[102] P. Kumar and H. H. Huang, "G-Store: High-performance graph store for trillion-edge processing," in *Proc. Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2016, pp. 830–841.

[103] P. Kumar and H. H. Huang, "Graphone: A data store for real-time analytics on evolving graphs," in *Proc. 17th USENIX Conf. File Storage Technol.*, 2019, pp. 249–263.

[104] G. Kwasniewski, M. Kabić, M. Besta, J. VandeVondele, R. Solcà, and T. Hoefler, "Red-blue pebbling revisited: Near optimal parallel matrix-matrix multiplication," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2019, Art. no. 24.

[105] K. G. Larsen, J. Nelson, H. L. Nguyen, and M. Thorup, "Heavy hitters via cluster-preserving clustering," *Commun. ACM*, vol. 62, no. 8, pp. 95–100, 2019.

[106] M. M. Lee, I. Roy, A. AuYoung, V. Talwar, K. Jayaram, and Y. Zhou, "Views and transactional storage for large graphs," in *Proc. ACM/IFIP/USENIX Middleware*, 2013, pp. 287–306.

[107] H. Lin, X. Zhu, B. Yu, X. Tang, W. Xue, W. Chen, L. Zhang, T. Hoefler, X. Ma, X. Liu *et al.*, "ShenTu: Processing multi-trillion edge graphs on millions of cores in seconds," in *Proc. Int. Conf. High-Perform. Comput., Netw., Storage Anal.*, 2018, Art. no. 56.

[108] H. Liu and H. H. Huang, "Graphene: Fine-grained IO management for graph computing," in *Proc. USENIX Conf. File Storage Technol*, 2017, pp. 285–299.

[109] Y. Lu, G. Chen, B. Li, K. Tan, Y. Xiong, P. Cheng, J. Zhang, E. Chen, and T. Moscibroda, "Multi-path transport for {RDMA} in datacenters," in *Proc. Symp. Netw. Syst. Des. Implementation*, 2018, pp. 357–371.

[110] S. Maass, C. Min, S. Kashyap, W. Kang, M. Kumar, and T. Kim, "Mosaic: Processing a trillion-edge graph on a single machine," in *Proc. 12th Eur. Conf. Comput. Syst.*, 2017, pp. 527–543.

[111] P. Macko, V. J. Marathe, D. W. Margo, and M. I. Seltzer, "Llama: Efficient graph analytics using large multiversioned arrays," in *Proc. IEEE 31st Int. Conf. Data Eng.*, 2015, pp. 363–374.

[112] M. Mariappan, J. Che, and K. Vora, "DZiG: Sparsity-aware incremental processing of streaming graphs," in *Proc. Eur. Conf. Comput. Syst.*, 2021, pp. 83–98.

[113] M. Mariappan and K. Vora, "Graphbolt: Dependency-driven synchronous processing of streaming graphs," in *Proc. ACM Eur. Conf. Comput. Syst.*, 2019, pp. 1–16.

[114] R. R. McCune *et al.*, "Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing," *ACM Comput. Surv.*, vol. 48, no. 2, pp. 1–39, 2015.

[115] A. McGregor, "Graph stream algorithms: A survey," *Proc. ACM SIGMOD Int. Conf. Manage. Data Record*, vol. 43, no. 1, pp. 9–20, 2014.

[116] F. McSherry, D. G. Murray, R. Isaacs, and M. Isard, "Differential dataflow," in *Proc. Conf. Innov. Data Syst. Res.*, 2013.

[117] G. T. Minton and E. Price, "Improved concentration bounds for count-sketch," in *Proc. Annual ACM-SIAM Symp. Discrete Algorithms*, 2014, pp. 669–686.

[118] V. Z. Moffitt and J. Stoyanovich, "Temporal graph algebra," in *Proc. 16th Int. Symp. Database Program. Lang.*, 2017, pp. 1–12.

[119] J. Mondal and A. Deshpande, "Managing large dynamic graphs efficiently," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2012, pp. 145–146.

[120] D. G. Murray, F. McSherry, M. Isard, R. Isaacs, P. Barham, and M. Abadi, "Incremental, iterative data processing with timely dataflow," *Commun. ACM*, vol. 59, no. 10, pp. 75–83, 2016.

[121] S. Muthukrishnan *et al.*, *Data Streams: Algorithms and Applications*. Hanover, MA, USA: Now Pub., 2005.

[122] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, "GraphPIM: Enabling instruction-level PIM offloading in graph computing frameworks," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2017, pp. 457–468.

[123] S. Neuendorffer and K. Vissers, "Streaming systems in FPGAs," in *Proc. Int. Workshop Embedded Comput. Syst.*, 2008, pp. 147–156.

[124] T. C. O'connell, "A survey of graph algorithms under extended streaming models of computation," in *Fundam. Problems Comput.*, Berlin, Germany: Springer, 2009, pp. 455–476.

[125] P. Peng and C. Sohler, "Estimating graph parameters from random order streams," 2018, *arXiv:1711.04881v1*.

[126] J. Pérez, M. Arenas, and C. Gutierrez, "Semantics and complexity of SPARQL," *ACM Trans. Database Syst.*, vol. 34, no. 3, 2009, Art. no. 16.

[127] V. Prabhakaran, M. Wu, X. Weng, F. McSherry, L. Zhou, and M. Haradasan, "Managing large graphs on multi-cores with graph awareness," in *Proc. USENIX Anuu. Tech. Conf.*, 2012, pp. 41–52.

[128] I. Robinson, J. Webber, and E. Eifrem, "Graph database internals," *Graph Databases*, 2nd Ed., chapter 7, Newton, MA, USA: O'Relly, 2015, pp. 149–170.

[129] M. A. Rodriguez, "The gremlin graph traversal machine and language (invited talk)," in *Proc. ACM 15th Symp. Database Program. Lang.*, 2015, pp. 1–10.

[130] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel, "Chaos: Scale-out graph processing from secondary storage," in *Proc. 25th Symp. Oper. Syst. Princ.*, 2015, pp. 410–424,.

[131] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: Edge-centric graph processing using streaming partitions," *ACM Symp. Oper. Syst. Princ*, 2013, pp. 472–488.

[132] P. Schmid, M. Besta, and T. Hoefler, "High-performance distributed RMA locks," in *Proc. 24th Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2016, pp. 19–30.

[133] H. Schweizer, M. Besta, and T. Hoefler, "Evaluating the cost of atomic operations on modern architectures," in *Proc. IEEE Int. Conf. Parallel Archit. Compilation*, 2015, pp. 445–456.

[134] D. Sengupta and S. L. Song, "Evograph: On-the-fly efficient mining of evolving graphs on GPU," in *Proc. Int. Supercomput. Conf.*, 2017, pp. 97–119.

[135] D. Sengupta *et al.*, "Graphin: An online high performance incremental graph processing framework," in *Proc. Eur. Conf. Parallel Process.*, 2016, pp. 319–333.

[136] M. Sha, Y. Li, B. He, and K.-L. Tan, "Accelerating dynamic graph analytics on GPUs.," in *Proc. VLDB Endowment*, vol. 11, no. 1, pp. 107–120, 2017.

[137] B. Shao, H. Wang, and Y. Li, "Trinity: A distributed graph engine on a memory cloud," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2013, pp. 505–516.

[138] F. Sheng, Q. Cao, H. Cai, J. Yao, and C. Xie, “GraPU: Accelerate streaming graph analysis through preprocessing buffered updates,” in *Proc. ACM Symp. Cloud Comput.*, 2018, pp. 301–312.

[139] F. Sheng, Q. Cao, and J. Yao, “Exploiting buffered updates for fast streaming graph analysis,” *IEEE Trans. Comput.*, vol. 70, no. 2, pp. 255–269, Feb. 2021.

[140] X. Shi, B. Cui, Y. Shao, and Y. Tong, “Tornado: A system for real-time iterative analysis over evolving data,” in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2016, pp. 417–430.

[141] X. Shi *et al.*, “Graph processing on GPUs: A survey,” *ACM Comput. Surv.*, vol. 50, no. 6, 2018, Art. no. 81.

[142] J. Shun and G. E. Blelloch, “Ligra: A lightweight graph processing framework for shared memory,” in *Proc. ACM Sigplan Notices*, vol. 48, pp. 135–146, 2013.

[143] N. Simsiri, K. Tangwongsan, S. Tirthapura, and K. Wu, “Work-efficient parallel union-find with applications to incremental graph connectivity,” in *Proc. Euro. Conf. Parallel Process.*, 2016, pp. 561–573.

[144] T. Suzumura, S. Nishii, and M. Ganse, “Towards large-scale graph stream processing platform,” in *Proc. ACM World Wide Web*, 2014, pp. 1321–1326.

[145] G. Szárnyas *et al.*, “An early look at the LDBC social network benchmark’s business intelligence workload,” in *Proc. 1st ACM SIGMOD Joint Int. Workshop Graph Data Manage. Experiences Syst. (GRADES) Netw. Data Anal.*, 2018, pp. 9:1–9:11.

[146] A. Tate *et al.*, “Programming abstractions for data locality,” in *Proc. Workshop Program. Abstr. Data Locality*, 2014.

[147] T. Tseng *et al.*, “Batch-parallel euler tour trees.,” in *Proc. SIAM Symp. Algorithm Eng. Experiments*, 2019, pp. 92–106.

[148] L. G. Valiant, “A bridging model for parallel computation,” *Commun. ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[149] J. van den Brand and D. Nanongkai, “Dynamic approximate shortest paths and beyond: Subquadratic and worst-case update time,” *IEEE Annu. Symp. Found. Comput. Sci.*, 2019, pp. 436–455.

[150] L. M. Vaquero, F. Cuadrado, and M. Ripeanu, “Systems for near real-time analysis of large-scale dynamic graphs,” 2014, *arXiv:1410.1903*.

[151] K. Vora *et al.*, “Aspire: Exploiting asynchronous parallelism in iterative algorithms using a relaxed consistency based DSM,” *ACM SIGPLAN Notices*, vol. 49, no. 10, pp. 861–878, 2014.

[152] K. Vora *et al.*, “KickStarter: Fast and accurate computations on streaming graphs via trimmed approximations,” *ACM SIGOPS Oper. Systems Rev.*, vol. 51, no. 2, pp. 237–251, 2017.

[153] K. Wang, Z. Zuo, J. Thorpe, T. Q. Nguyen, and G. H. Xu, “Rstream: Marrying relational algebra with streaming for efficient graph mining on a single machine,” in *Proc. USENIX Symp. Operating Syst. Des. Implementation*, 2018, pp. 763–782.

[154] M. Winter *et al.*, “Autonomous, independent management of dynamic graphs on GPUs,” in *Proc. IEEE High Perform. Extreme Comput. Conf.*, 2017, pp. 1–7.

[155] M. Winter, D. Mlakar, R. Zayer, H.-P. Seidel, and M. Steinberger, “faimGraph: High performance management of fully-dynamic graphs under tight memory constraints on the GPU,” in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2018, pp. 1–13.

[156] H. Wu, J. Cheng, S. Huang, Y. Ke, Y. Lu, and Y. Xu, “Path problems in temporal graphs,” in *Proc. VLDB Endowment*, vol. 7, no. 9, pp. 721–732, 2014.

[157] H. Wu, Y. Huang, J. Cheng, J. Li, and Y. Ke, “Reachability and time-based path queries in temporal graphs,” in *Proc. IEEE 32nd Int. Conf. Data Eng.*, 2016, pp. 145–156.

[158] M. Wu *et al.*, “Gram: Scaling graph computation to the trillions,” in *Proc. 6th ACM Symp. Cloud Comput.*, 2015, pp. 408–421.

[159] L. Xiangyu, L. Yingxiao, G. Xiaolin, and Y. Zhenhua, “An efficient snapshot strategy for dynamic graph storage systems to support historical queries,” *IEEE Access*, vol. 8, pp. 90838–90846, 2020.

[160] W. Xie, Y. Tian, Y. Sismanis, A. Balmin, and P. J. Haas, “Dynamic interaction graphs with probabilistic edge decay,” in *Proc. IEEE 32nd Int. Conf. Data Eng.*, 2015, pp. 1143–1154.

[161] M. Zaharia *et al.*, “Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing,” in *Proc. USENIX Proc. Symp. Netw. Syst. Des. Implementation*, 2012, pp. 15–28.

[162] M. Zaharia *et al.*, “Apache spark: A unified engine for big data processing,” *Commun. ACM*, vol. 59, no. 11, pp. 56–65, 2016.

[163] A. Zaki, M. Attia, D. Hegazy, and S. Amin, “Comprehensive survey on dynamic graph models,” *Int. J. Adv. Comput. Sci. Appl.*, vol. 7, no. 2, pp. 573–582, 2016.

[164] J. Zhang, “A survey on streaming algorithms for massive graphs,” *Managing Mining Graph Data*, 2010, pp. 393–420.

[165] S. Zhou, R. Kannan, H. Zeng, and V. K. Prasanna, “An FPGA framework for edge-centric graph processing,” in *Proc. 15th ACM Int. Conf. Comput. Front.*, 2018, pp. 69–77.

[166] X. Zhu, G. Feng, M. Serafini, X. Ma, J. Yu, L. Xie, A. Aboulnaga, and W. Chen, “LiveGraph: A transactional graph storage system with purely sequential adjacency list scans,” 2019, *arXiv:1910.05773*.

Maciej Besta is currently a researcher with ETH Zurich. His research focuses on understanding and accelerating large-scale irregular graph processing in any types of settings and workloads.

Marc Fischer is currently a software developer and consultant with PRODYNA (Schweiz) AG. His research interests include large-scale graph databases, data modeling, use-case analysis, and developing custom solutions for a broad range of business applications.

Vasiliki Kalavri is currently an assistant professor with the Department of Computer Science, Boston University. Her research research interests include distributed stream processing and large-scale graph analytics.

Michael Kapralov is currently an assistant professor with the School of Computer and Communication Sciences, EPFL, and part of the EPFL Theory Group. He works on theoretical foundations of big data analysis.

Torsten Hoefler is currently a professor with ETH Zurich, where he leads the Scalable Parallel Computing Lab. His research focuses on understanding performance of parallel computing systems ranging from parallel computer architecture through parallel programming to parallel algorithms.