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

Maciej Besta[1], Marc Fischer[2], Vasiliki Kalavri[3], Michael Kapralov[4], Torsten Hoefler[1]
[1]Department of Computer Science, ETH Zurich
[2]PRODYNA (Schweiz) AG; [3]Department of Computer Science, Boston University
[3]School of Computer and Communication Sciences, EPFL

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 parallel execution of graph updates, 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 parallelism, 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 analyzing 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.

1 Introduction

Analyzing massive graphs has become an important task. Example applications are investigating the Internet structure [42], analyzing social or neural relationships [24], or capturing the behavior of proteins [65]. Efficient processing of such graphs is challenging. First, these graphs are large, reaching even tens of trillions of edges [51], [142]. 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 [14].

Graph streaming frameworks such as STINGER [76] or Aspen [63] emerged to enable processing and analyzing dynamically evolving graphs. Contrarily to static frameworks such as Ligra [190], [98], 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 in parallel 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.

Moreover, we crystallize the meaning of different concepts in streaming and dynamic graph processing. We investigate the notions of *temporal*, *time-evolving*, *online*, and *dynamic* graphs. We also discuss the differences between graph streaming frameworks and the *edge-centric* engines, as well as 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* [154] or *dynamic graphs* [40] that come with different goals and techniques. Moreover, each of these settings comes with different *models*, for example the *dynamic graph stream* model [119] or the *semi-streaming* model [75]. 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 [210]. These models are 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 [48] or Differential Dataflow [156]. 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, parallel execution, data distribution, targeted architecture, and others.

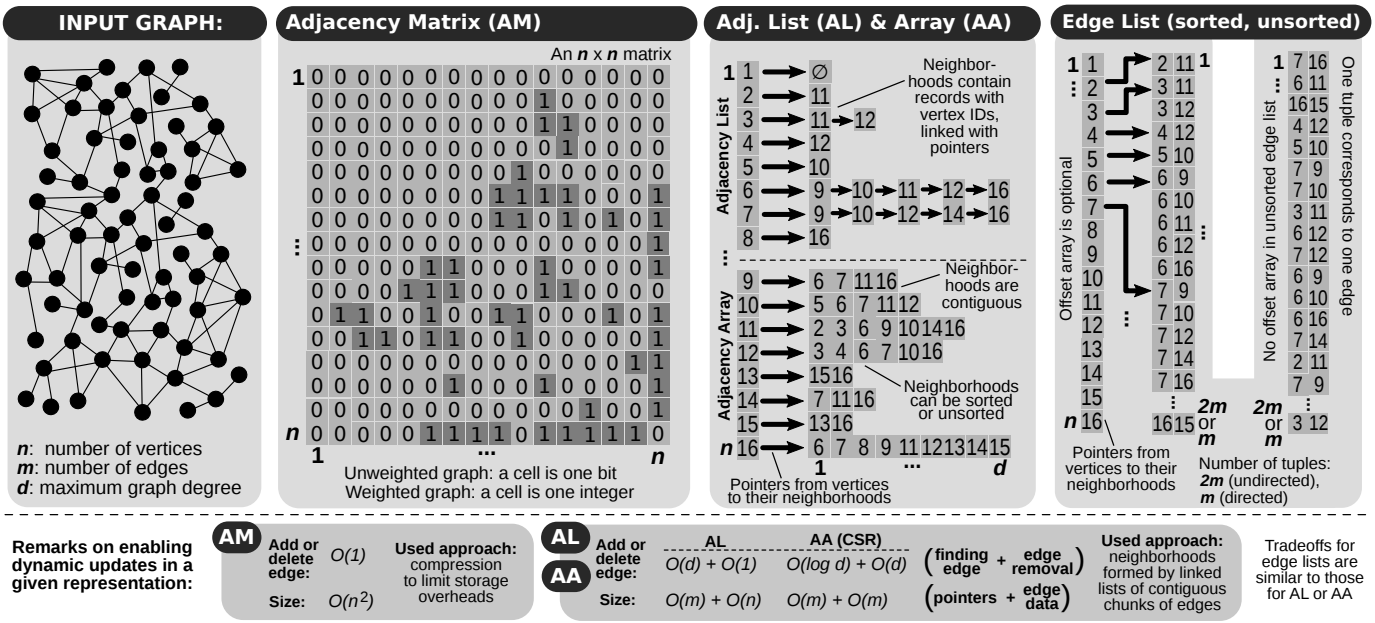

Fig. 1: Illustration of fundamental graph representations (Adjacency Matrix, Adjacency List, Edge List) and remarks on their usage in dynamic settings.

- 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 [154], [213], [7], [169], analyses on *static* graph processing [100], [67], [188], [22], [153], [36], and on *general* streaming [118]. Finally, only one prior work summarized types of graph updates, partitioning of dynamic graphs, and some challenges [204].

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); $V = \{v_1, ..., v_n\}$ is a set of vertices and $E = \{e_1, ..., e_m\} \subseteq V \times V$ 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 $e \in E$.

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), and the *edge list* (EL), and the Adjacency Array (AA, aka CSR). We illustrate these representations and we provide remarks on their dynamic variants in Figure 1. In **AM**, a matrix $\mathbf{M} \in \{0,1\}^{n,n}$ determines the connectivity of vertices: $\mathbf{M}_{u,v} = 1 \Leftrightarrow (u, v) \in E$. 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 $v \in A_u \Leftrightarrow (u, v) \in E$. 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 $\mathcal{O}(|A_u|) \subseteq \mathcal{O}(d)$ 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 [39]. Finally, **Adjacency Array (AA)** (aka **CSR**) resembles AL but it consists of n *contiguous arrays* with neighborhoods of vertices. Each array is usually sorted by vertex IDs. AA 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 performs some computation on a graph and it *returns information* about the graph; the information can be simple (e.g., the degree of a given vertex) or complex (e.g., some subgraph). A graph update *modifies* the graph itself, 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 Figure 2. To foster developing more powerful and versatile systems for dynamic and streaming graph processing, we outline both applied and theoretical concepts.

3.1 Applied Dynamic and Streaming Graph Processing

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

3.1.1 Streaming, Dynamic, and Time-Evolving Graphs

Many works [71], [63] use a term "streaming" or "streaming graphs" to refer to a setting in which a graph is *dynamic* [185] (also referred to as *time-evolving* [111] or *online* [77]) and it can be modified with updates such as edge insertions or deletions. This setting is the primary focus of this survey.

3.1.2 Graph Databases and NoSQL Stores

Graph databases [35] are related to streaming and dynamic graph processing in that they support graph updates. Graph databases (both "native" graph database systems and

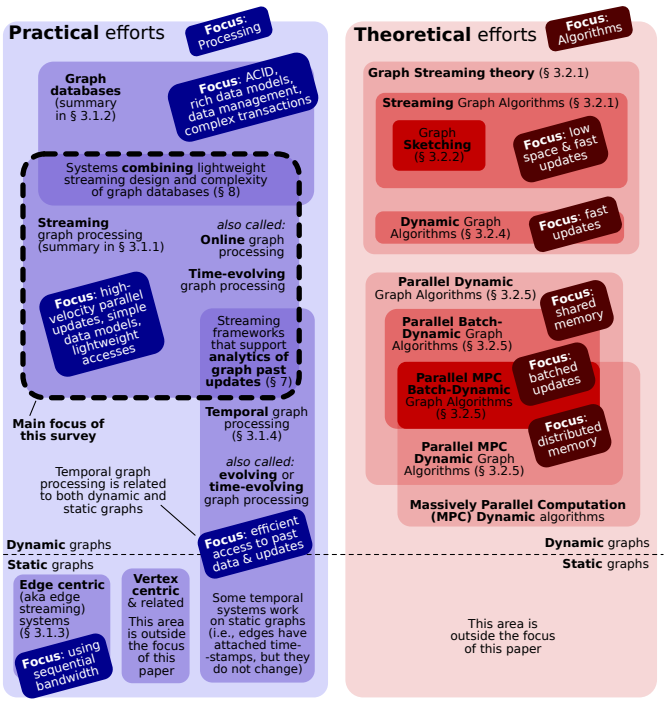

Fig. 2: Overview of the domains and concepts in the practice and theory of streaming and dynamic graph processing and algorithms. This work focuses on streaming graph processing and its relations to other domains.

NoSQL stores used as graph databases (e.g., RDF stores or document stores)) were described in detail in a recent work [35] 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 [215], [38], [167], [179] 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 [179], ShenTu [142], and several FPGA designs [38]. Such designs are outside the main focus of this survey; some of them were described by other works dedicated to static graph processing [38], [67].

3.1.4 Temporal (or Time-Evolving) Graph Processing

There exist efforts into analyzing *temporal* (also referred to – somewhat confusingly – as *[time]-evolving*) graphs [205], [198], [175], [158], [157], [102], [101], [82], [129], [128]. As noted by Dhulipala et al. [63], 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. Any additional snapshots are mainly dedicated to more efficient ingestion of graph updates, and *not* to preserving historical data for time-related analytics. Moreover, temporal graphs are *not necessarily dynamic* and do not have to appear as a stream. What is important is that edges and/or vertices have attached timestamps that enable temporal analysis.

These efforts are outside the focus of this survey. Still, *we describe concepts and systems that – while focusing on high-performance maintenance and analytics of a dynamic graph in its recent state – also enable keeping and processing historical data.*

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* [75], [57], 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) [41]. Different streaming models can introduce additional assumptions, for example the Sliding Window Model provides restrictions on the *length of the stream* [57]. 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., [138] and [122] 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 [75]), 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 [120], [73], [44], [21], [20], [171], [121]. The reader is referred to surveys on graph streaming algorithms [164], [95], [155] for a more complete set of references.

Applicability in Practical Settings Streaming algorithms can be used when there are hard limits on the maximum space allowed for keeping the processed graph, as well as a need for very fast updates per edge. Moreover, one should bear in mind that many of these algorithms provide approximate outcomes. Finally, the majority of these algorithms assumes the knowledge of certain structural graph properties in advance, most often the number of vertices n.

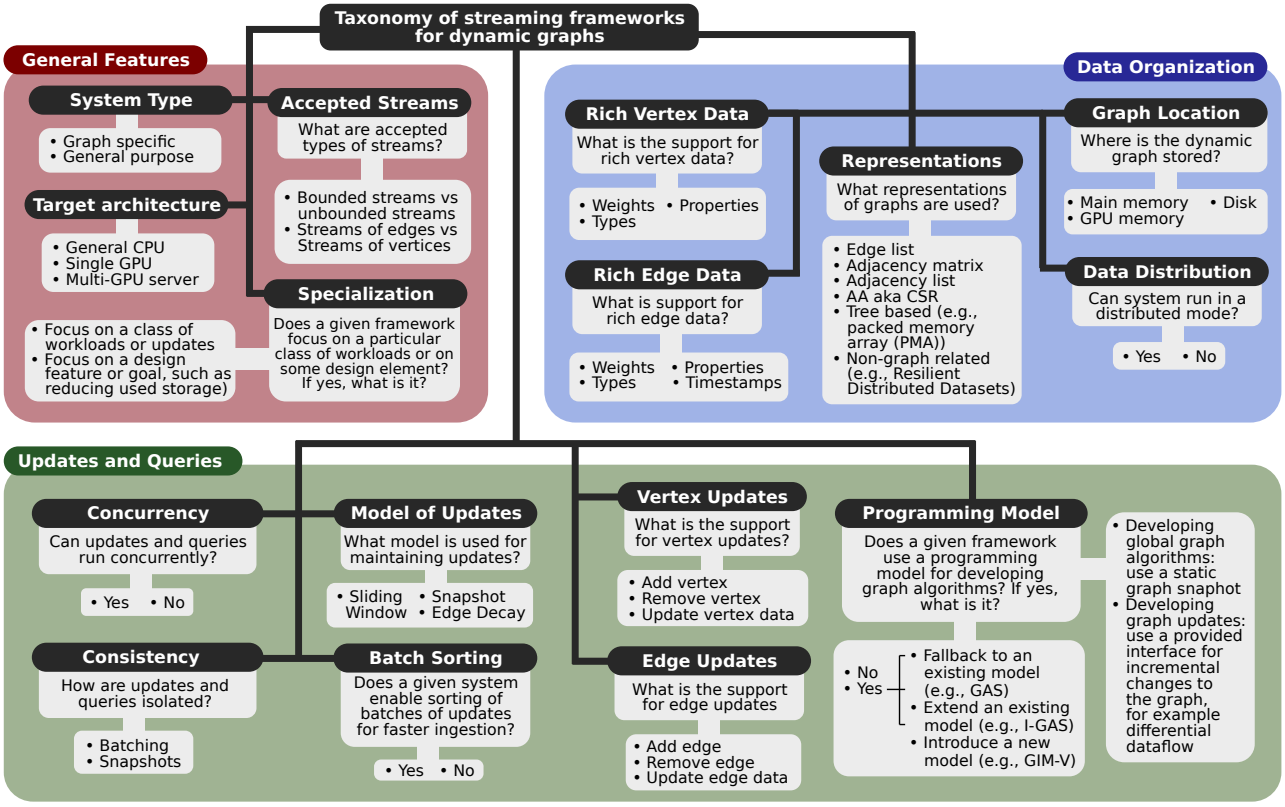

Fig. 3: Overview of the identified taxonomy of dynamic graph streaming frameworks

3.2.2 Graph Sketching and Dynamic Graph Streams

Graph sketching [12] is an influential technique for processing graph streams with both insertions and deletions. The idea is to apply classical sketching techniques such as CountSketch [159] or distinct elements sketch (e.g., HyperLogLog [80]) to the edge incidence matrix of the input graph. Existing results show how to approximate the connectivity and cut structure [12], [16], spectral structure [123], [122], shortest path metric [12], [124], or subgraph counts [119], [117] using small sketches. Extensions to some of these techniques to hypergraphs were also proposed [96].

3.2.3 Multi-Pass Streaming Graph Algorithms

Some streaming graph algorithms use the notion of a *bounded stream*, i.e., the number of graph updates is bounded. Streaming and applying all such updates once is referred to as a single *pass*. Now, some streaming graph algorithms allow for *multiple passes*, i.e., streaming all edge updates *more than once*. This is often used to improve the approximation quality of the computed solution.

There exist numerous other works in the theory of streaming graphs. Variations of the semi-streaming model allow stream manipulations across passes, (also known as the *W-Stream* model [61]) or stream sorting passes (known as the *Stream-Sort* model [8]). We omit these efforts are they are outside the scope of this paper.

3.2.4 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 [40], [49], [23], [202], [70], [81], [68] and references within for some of the most recent developments.

Applicability in Practical Settings Dynamic graph algorithms can match settings where primary focus is on fast updates, without severe limitations on the available space.

3.2.5 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 [5], [191], [4], [200]. 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*** [69] that *combines the parallel batch-dynamic model with the Massively Parallel Computation (MPC) model* [125] was also recently described. The MPC model is motivated by distributed frameworks such as MapReduce [60]. 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 [125].

Finally, ***another variant*** of the MPC model that addresses dynamic graph algorithms *but without considering batches*, was also recently developed [108].

Applicability in Practical Settings Algorithms developed in the above models may be well-suited for enhancing streaming graph frameworks as these algorithms explicitly (1) maximize the amount of parallelism by using the concept of batches, and (2) minimize time to ingest updates.

4 Taxonomy of Frameworks

We now identify a taxonomy of graph streaming frameworks, see Figure 3 for a summary. Tables 1–2 provide information on concrete frameworks. The identified taxonomy aspects divide into three classes: *general features*, *data organization*, and *updates and queries*. **The first class** groups aspects of the general system purpose and setting: the targeted architecture, whether a system is general-purpose or specifically targeting graph workloads, what types of streams a system accepts, and whether a system puts special focus on a particular class of graph workloads and/or any other feature. **The second class** describes design aspects related to the used data structures: used graph representation(s), support for data distribution, the location of maintained graph (e.g., main memory or GPU storage), and supported "rich" edge and vertex data types such as attached values or properties. **The third class** groups design aspects related to graph updates and queries: supported edge and vertex updates, a form of applying graph updates to the maintained graph (e.g., updates are ingested in batches or one at a time), support for concurrent execution of both updates and queries, the used programming model, and the used model for maintaining graph updates,

4.1 Accepted Types of Streams

We also identify different *types of streams*. First, streams can have different **lengths**: they can be **bounded** (i.e., have predetermined length) or **unbounded** (i.e., there is bound on the number of incoming updates). Second, a stream can be represented either as a sequence of edges **(edge stream)** or as a sequence of vertices with their adjacency lists **(vertex stream)** [2]. In the following, unless specified otherwise, we assume unbounded edge streams, which dominate the landscape of streaming and dynamic graph processing.

4.2 Architectural Aspects

We also consider the **targeted architecture** of a given system, the **location of the maintained graph data** (e.g., main memory or GPU memory), or whether a system is **general-purpose** or is it developed specifically for **graph analytics**.

4.3 Supported Types of Vertex and Edge Data

Contrarily to graph databases that heavily use rich graph models such as Labeled Property Graph [17], graph streaming frameworks usually offer simple data models, focusing on the *graph structure* and not on *rich data attached to vertices or edges*. Still, different frameworks support basic additional vertex or edge data, most often **weights**. Moreover, in certain systems, both an edge and a vertex can have a **type** or an **attached property (value)**. 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.

4.4 Used Graph Representations

We also investigate how different streaming frameworks represent the maintained graph. Some frameworks use one of the **fundamental graph representations** (AL, EL, CSR, or AM) which are described in Section 2. For example, Kineograph [50] uses AL. No systems that we analyzed use an *uncompressed* AM as it is inefficient with $\mathcal{O}(n^2)$ space, especially for sparse graphs. Systems that use AM, for example GraphIn, focus on compression of the adjacency matrix [32], trying to mitigate storage and query overheads.

Most frameworks **combine** these fundamental representations. For example, STINGER combines **AL** and **CSR** by dividing each neighborhood into contiguous *chunks* that are larger than a single vertex ID (as in basic AL) but smaller than a whole neighborhood (as in basic CSR). This offers a tradeoff between flexible modifications in AL and efficient neighborhood traversals in CSR [177].

A few systems use graph representations **based on trees**. For example, Sha et al. [185] use a variant of *packed memory array* (PMA), which is an array with all neighborhoods, augmented with an implicit binary tree structure that enables edge insertions and deletions in $O(\log^2 n)$ time.

We also consider whether a framework supports **data distribution** over multiple serves. Any of the above representations can be developed for either a single server or for a distributed-memory setting. Details of such distributed designs are system-specific.

Finally, frameworks constructed on top of more general infrastructure use a representation provided by the underlying system. For example, GraphTau [111], built on top of Apache Spark [212], uses the underlying data structure called Resilient Distributed Datasets (RDDs) [212].

4.5 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.

4.6 Architecture of Applying Graph Updates

A **form of incorporating updates** into the graph structure determines the *architecture* of a given streaming framework, and it dictates the *amount of parallelism* that can be achieved by a given framework. In this context, we identify two key forms of applying graph updates. First, as also noted by Dhulipala et al. [63], a system can **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 amount of parallelism exploiting temporal and spatial locality when digesting updates as it does not have to resolve the problem of the consistency of graph queries running interleaved, in parallel, with updates being digested. However, it does *not* enable a concurrent execution of updates and queries. Second, a system can enable to execute updates and queries concurrently. A popular form of implementing this is based on **snapshots**. Here, updates and queries are isolated from each other by making queries execute on a snapshot of the graph data. Depending on a

system, the number of snapshots, the scope of data duplication (i.e., only a part of the graph may be copied into a new snapshot), and the policy of merging snapshots may differ. Finally, there are **system-specific** schemes that do not rely on snapshots. An important example is **Differential Dataflow** [156], 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.

Multiple streaming frameworks use an 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 parallelization of the ingestion of updates*: updates associated with different vertices can often be straightforwardly incorporated *in parallel*.

4.7 Model of Maintaining Graph Updates

The ingestion of updates into the maintained graph (and the associated modification of maintained structural graph properties) can be done using different approaches.

Most systems are based on a **"snapshot" model**[1], 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 $e \in E$ [210].

Some streaming systems use the **sliding window model**, in which edges beyond certain timestamps 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 [210].

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 [210]. 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**.

[1]Here, the word "snapshot" means "a complete view of the graph, with all its updates". The naming in the literature is somewhat confusing in this respect, as "snapshot" can also mean "a specific copy of the graph generated for more efficient parallel processing of updates and queries", cf. § 4.6.

4.8 Provided Programming Model

Developing algorithms for deriving structural properties of dynamic graphs can be facilitated by a dedicated *programming model*. As of now, there are no *established* programming models for dynamic graph analysis. Some frameworks, for example GraphInc [46], **fall back** to a model used for static graph processing (most often the vertex-centric model [148], [116]), and make the dynamic nature of the graph transparent to the developer. Another recent example is GraphBolt [149] that offers the Bulk Synchronous Parallel (BSP) [201] programming model and combines it with incremental updates to be able to solve certain graph problems on dynamic graphs. Other engines **extend** an existing model for static graph processing. For example, GraphIn [184] extends the gather-apply-scatter (GAS) paradigm [143] to enable reacting to incremental updates. Finally, certain systems offer a **novel** model for harnessing the dynamic nature of streaming graphs. An example is Tegra [110], a recent design that offers a Timelapse abstraction and an ICE model that, together, enable retrieving past graph snapshots and using them when deriving different structural graph properties.

Systems such as Kineograph [50], CellIQ [109], and Chronos [101] operate on a **snapshot-based model**. They use time discretization to express an evolving graph G_S as a series of *static* graphs $G_1, G_2, \ldots$, each of which represents a snapshot of G_S at a particular point in time. Computations are defined via a BSP model which operates independently per snapshot. In some cases, cross-snapshot analytics, such as sliding window computations (cf. § 4.7) are also supported. Snapshots are created by either making a copy of the entire graph periodically or by maintaining a graph data structure that can be updated incrementally to reflect a state of the graph at a given point in time.

5 Frameworks

We now analyze and discuss selected frameworks that maintain *dynamic* graphs. We select systems for more detailed descriptions to cover a wide spectrum of supported updates, used data representations, offered programming models, and utilized models for the maintenance of graph updates. In these descriptions, we focus on key design details related to ***data organization*** and ***parallelism***; these aspects usually constitute core elements of the *architecture* of a given system. Tables 1–2 illustrate the details of different graph streaming systems, including the ones described in this section. The tables show features supported by considered systems. 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. "❓" indicates we were unable to infer this information based on the available documentation[2].

5.1 STINGER [71] And Its Variants

STINGER [71] is a data structure and a corresponding framework. It adapts and extends the *CSR format* to support graph updates. Contrarily to the static CSR design, where IDs of the neighbors of a given vertex are stored contiguously, neighbor IDs in STINGER are divided into contiguous

[2]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.

Reference	D?	C?	Edge updates	Vertex updates	Data store	Targeted architecture	Remarks
GRAPH-SPECIFIC STREAMING SYSTEMS							
STINGER [71]	👎	👎	👍 (A/R)	👍* (A/R)	Main memory	General CPU	*Vertex removal is unclear
GraphInc [46]	👎	👎	👍 (A/R/U)	👍 (A/R/U)	?	General CPU	Extends Apache Giraph [1]
Kineograph [50]	👍	👍	👍 (A/custom)	👍 (A/custom)	?	General CPU	Custom update functions are possible
Mondal et al. [160]	👍	👍**	👍* (A)	👍* (A)	Main memory	General CPU	*Details are unclear, **no consistency guarantee
Concerto [139]	👍	👍	👍 (A/U)	👍 (A/U)	Main memory	General CPU	—
UNICORN [195]	👍	👎	👍 (A/R)	👍 (A/R)	Main memory	General CPU	—
LLAMA [146]	👎	👍	👍 (A/R)	👍 (A/R)	Main memory or disk	General CPU	—
CellIQ [109]	👍	👍	👍 (A/R)	👍 (A/R)	Disk (HDFS)	General CPU	—
DISTINGER [76]	👍	👎	👍 (A/R)	👍 (A/R)	Main memory	General CPU	—
cuSTINGER [93]	👎	👎	👍 (A/R)	👍 (A/R)	GPU memory	Single GPU	—
GraphTau [111]	👍	👍	👍 (A/R)	👍 (A/R)	Main memory*	General CPU	*Data may be placed in an underlying graph database
GraphIn [184]	👎*	👎	👍* (A/R)	👍* (A/R)	Main memory	General CPU	*Details are unclear
aimGraph [209]	👎	👎	👍 (A/R)	👎	GPU memory	Single GPU	—
Sha et al. [185]	👍*	👎	👍 (A/R)	👎	GPU memory	Multi-GPU server	*The input graph can be distributed over multiple GPUs **on one server**.
EvoGraph [183]	👎	👎	👍 (A/R)	👍 (A/R)	Main memory	Single GPU	Supports multi-tenancy to share GPU resources
KickStarter [206]	👍	👎	👍 (A/R)	?	Main memory	General CPU	—
ZipG [127]	👍	👍	👍 (A/R/U)	👍 (A/R/U)	Main memory	General CPU	Limited support for computations on windows
Hornet [45]	👎	👍	👍 (A/R/U)	👍 (A/R/U)	GPU or main memory	Single GPU	Hornet [45] is platform independent, but presented for GPUs. It allows bulk updates
BiGJoin [15]	👍	👍	👍 (A)	👍 (A)	Main memory or disk	General CPU	—
Aspen [63]	👎	👍	👍 (A/R)	👍 (A/R)	Disk	General CPU	Scaling requires a shared address space
faimGraph [208]	👎	👎	👍 (A/R)	👍 (A/R)	GPU or main memory	Single GPU	—
GraphOne [135]	👎	👍	👍 (A/R)	👍 (A/R)	Main memory	General CPU	—
Tegra [110]	👍	👍	👍 (A/R)	👍 (A/R)	Main memory*	General CPU	*Data may be placed in an underlying graph database
LiveGraph [216]	👎	👍	👍 (A/R/U)	👍(A/R/U)	main memory	General CPU	—
GENERAL STREAMING SYSTEMS that support graph processing							
Apache Flink [48]	👍	👍	👍	👍	?	General CPU	Support for batch data processing
Naiad [163]	👍	👍	👍	👍	?	General CPU	Uses **differential dataflow** model
Tornado/Storm [187]	👍	👍	👍	👍	?	General CPU	Update capabilities are not explicitly mentioned

TABLE 1: Summary of the features of selected representative works sorted by publication date. "D" (distributed): A design targets distributed environments such as clusters, supercomputers, or data centers. "C" (concurrent updates and queries): A design supports updates (e.g., edge insertions and removals) proceeding concurrently with queries that access the graph structure. "A": add, "R": remove, "U": update. "👍": A design offers a given feature. "👍": A design offers a given feature in a limited way. "👎": A design does not offer a given feature. "?": Unknown.

blocks of a pre-selected size. These blocks form a *linked list*. The block size is identical for all the blocks except for the last blocks in each list. One neighbor vertex ID u in the neighborhood of a vertex v corresponds to one edge (v, u). STINGER supports both vertices and edges with different *types*. One vertex can have adjacent edges of different types. One block always contains edges of one type only. Besides the associated neighbor vertex ID and type, each edge has its weight and two time stamps. In addition to this, each edge block contains certain metadata, for example lowest and highest time stamps in a given block. Moreover, STINGER provides the edge type array (ETA) *index data structure*. ETA contains pointers to all blocks with edges of a given type.

To increase *parallelism*, STINGER updates a graph in *batches*. For graphs that are not scale-free, a batch of around 100,000 updates is first sorted so that updates to different vertices are grouped. In the process, deletions may be separated from insertions (they can also be processed concurrently with insertions). For scale-free graphs, there is no sorting phase and each update is processed in parallel. *Fine locking* on single edges is used for synchronization of updates to the neighborhood of the same vertex. To insert an edge or to verify if an edge exists, one traverses a selected list of blocks, taking $O(d)$ time. Consequently, inserting an edge into N_v takes $O(d_v)$ work and depth. STINGER is optimized for the Cray XMT supercomputing systems that allow for massive thread-level parallelism. Still, it can also be executed on general multi-core commodity servers.

DISTINGER [76] is a distributed version of STINGER that targets *"shared-nothing" commodity clusters*. DISTINGER inherits the STINGER design, with the following modifications. First, a designated *master* process is used to interact between the DISTINGER instance and the outside world. The master process maps external (application-level) vertex IDs to the internal IDs used by DISTINGER. The master process maintains a list of *slave* processes and it assigns incoming queries and updates to slaves that maintain the associated part of the processed graph. Each slave maintains and is responsible for updating a portion of the vertices together with edges attached to each of these vertices. The graph is partitioned with a simple hash-based scheme. The inter-process communication uses MPI [87], [94] with established optimizations such as message batching or overlap of computation and communication.

cuSTINGER [93] extends STINGER for CUDA GPUs. The main design change is to replace lists of edge blocks with *contiguous adjacency arrays*. Moreover, contrarily to STINGER, cuSTINGER *always separately processes updates and deletions*, to better utilize *massive parallelism in GPUs*. cuSTINGER offers several "meta-data modes": based on user needs, the framework can support only unweighted edges, weighted edges without any additional associated data, or edges with weights, types, and additional data such as time stamps. However, the paper focuses on unweighted graphs that do not use time stamps and types, and the exact GPU design of the last two modes is unclear [93].

5.2 LLAMA [146]

LLAMA [146] – similarly to STINGER – digests graph updates in batches. It differs from STINGER in that each such batch generates a new *snapshot* of graph data. Specifically, the graph in LLAMA is represented using a *multi-versioned extension of the adjacency list*. The adjacency list of a vertex can be divided into smaller parts, each part belonging to a different snapshot. Each such part is contiguous. For example, if there is a batch with edge insertions into the

Reference	Rich edge data	Rich vertex data	Tested workloads	Representation	Focus	Remarks, used programming models
GRAPH-SPECIFIC STREAMING SYSTEMS						
STINGER [71]	👍 (T, W, TS)	👍 (T)	Clustering, BC, BFS, CC, k-core	CSR+AL	Update time	—
GraphInc [46]	👍 (P)	👍 (P)	SP, CC, PageRank	?	Iterative algorithms	GraphInc uses the **Pregel programming model**. No performance data reported
Kineograph [50]	👎	👎	TunkRank, SP, K-exposure	AL	Fault tolerance	—
Mondal et al. [160]	👎	👎	—	?	Data replication and partitioning, load balancing	—
Concerto [139]	👍 (P)	👍 (P)	k-hop, k-core	AL	Scalability, transactions	—
UNICORN [195]	👎	👎	PageRank, RWR	?	Iterative algorithms	Introduces **a programming model called Incremental GIM-V**
LLAMA [146]	👍 (P)	👍 (P)	PR, BFS, TC	CSR	Out-of-memory analysis	—
CellIQ [109]	👍 (P)	👍 (P)	Handoff Sequences, Persistent Hotspots	Data stream	Cellular network analysis	The authors use the **Sliding Window Model**
DISTINGER [76]	👍 (T, W, TS)	👍 (T, W)	PR	CSR	Distributed processing	—
cuSTINGER [93]	👍* (W, P, TS)	👍* (W, P)	TC	AL	Update time, memory usage	*No details provided for properties
GraphTau [111]	👎	👎	PageRank, CC	RDDs	Fault tolerance	The authors use the **Sliding Window Model**
GraphIn [184]	👎	👍 (P)	BFS, CC, clustering coefficient	EL+CSR	Iterative algorithms	The authors also propose **a programming model called I-GAS**
aimGraph [209]	👍* (W)	👍* (W)	—	AL	Update time, memory usage	*No details are provided
Sha et al. [185]	👍 (TS)	👎	PR, BFS, CC	Tree based (PMA)	GPU-friendly, update time	The authors use the **Sliding Window Model**
EvoGraph [183]	👎	👎	TC, CC, BFS	EL+CSR	Iterative algorithms, GPU sharing	EvoGraph relies on the **I-GAS model** [184]
KickStarter [206]	👍 (W)	👎	SSWP, CC, SSSP, BFS	?	Iterative algorithms	Approach similar to Tornado [187]
ZipG [127]	👍 (T, P, TS)	👍 (P)	TAO, LinkBench	Compressed flat files	—	
Hornet [45]	👍 (W)	👎	BFS, SpMV, K-Truss	AL+CSR	Storage, update time, re-allocation	—
BiGJoin [15]	👎	👎	Graph pattern matching (triangles, 4-cliques, diamonds, 5-cliques, ...)	Data stream	Pattern matching	Restricted to equi-join algorithms
Aspen [63]	👎	👎*	BFS, BC, MIS, 2-hop, Local-Cluster	Tree + C-Trees	Space usage, data locality	*Support for weighted edges is possible
faimGraph [208]	👍 (W, P)	👍 (W, P)	PageRank, Static Triangle Counting	AL	Add/Remove, memory usage	—
GraphOne [135]	👍 (W)	👎	BFS, PageRank, 1-Hop-query	EL+AL	Stream and batch analytics	—
Tegra [110]	👍 (P)	👍 (P)	PageRank, CC	Tree	Window processing	The authors use the **Sliding Window Model**
LiveGraph [216]	👍 (T, P)	👍 (P)	TAO, LinkBench, PR, CC	Transactional Edge Log (AL)	Transactional management and graph analytics	—
GENERAL STREAMING SYSTEMS that support graph processing						
Apache Flink [48]	👍	👍	—	Data stream	Fault tolerance	Fault tolerance by checkpoints and recovery
Naiad [163]	👍	👍	PageRank, CC	Data stream	Work sharing, synchronization	Uses **differential dataflow** model
Tornado [187]	?	?	SSSP, PageRank	?	Iterative algorithms, fault tolerance	—

TABLE 2: Summary of the features of selected representative works sorted by publication date. "👍": A design offers a given feature. "👍": A design offers a given feature in a limited way. "👎": A design does not offer a given feature. "T": types, "P": properties, "W": weights, "TS": timestamps. "?": Unknown.

adjacency list of a vertex v, this batch may become a part of v's adjacency list within a new snapshot. Moreover, the LLAMA design features a structure that is shared by all snapshots, which maps vertices to per-vertex structures.

5.3 GraphIn [184]

GraphIn [184] uses a *hybrid* dynamic data structure. First, it uses an AM (in the *CSR format*) to store the adjacency data. This part is *static* and is not modified when updates arrive. Second, incremental graph updates are stored in dedicated *edge lists*. Every now and then, the AM with graph structure and the edge lists with updates are merged to update the structure of AM. Such a design maximizes performance and the amount of used parallelism when accessing the graph structure that is mostly stored in the CSR format.

5.4 GraphTau [111]

GraphTau [111] is a framework based on Apache Spark and its data model called resilient distributed datasets (RDD) [212]. RDDs are read-only, immutable, partitioned collections of data sets that can be modified by different operators (e.g., map, reduce, filter, and join). Similarly to GraphX [91], GraphTau exploits RDDs and stores a graph snapshot (called a GraphStream) using *two RDDs: an RDD for storing vertices and edges*. Due to the snapshots, the framework offers fault tolerance by replaying the processing of respective data streams. Different operations allow to receive data form multiple sources (including graph databases such as Neo4j and Titan) and to include unstructured and tabular data (e.g., from RDBMS). Further, GraphTau provides options to write custom iterative and window algorithms by defining a directed acyclic graph (DAG) of operations. The underlying Apache Spark framework analyzes the DAG and processes the data in parallel on a compute cluster. GraphTau only enables using the window sliding model. To maximize parallelism when ingesting updates, it applies the snapshot scheme: graph workloads run in parallel with graph updates using different snapshots.

5.5 faimGraph [208]

faimGraph [208] (*f*ully-dynamic, *a*utonomous, *i*ndependent *m*anagement of *graphs*) is a library for graph processing on a single GPU with focus on fully-dynamic edge and vertex updates (add, remove). It allocates a single block of memory on the GPU and uses a memory manager to autonomously handle data management without round-trips to the CPU. The design does not return free memory to the device, but keeps it allocated as it might be used during graph processing - so the parallel use of the GPU for other processing

is limited. In such cases, faimGraph can be reinitialized to claim memory (or expand memory if needed). The library implements the adjacency list. Vertices are stored in dedicated vertex *data blocks* that can also contain user-defined properties. Since the edge lists might grow and shrink, edges are stored in *pages that form a linked list*. Properties can be stored together with edges. Updates can be received from the device or from the host. Further, faimGraph relies on a bulk update scheme, where queries cannot be interleaved with updates. This supports exploiting parallelism of the GPU by running updates in parallel.

5.6 Hornet [45]

Hornet [45] is a data structure and associated system that focuses on efficient batch updates (inserting, deleting, and updating vertices and edges), and more effective memory utilization by requiring no re-allocation and no re-initialization of used data structures during computation. To achieve this, Hornet implements its own memory manager. The graph is maintained using an AL: vertices are stored in an array, with pointers pointing to the associated adjacency list. The lists are (transparently to the user) stored in *blocks* that can hold edges in counts that are powers of two. The allocation of specific edge lists to specific blocks is resolved by the system. Finally, B^+ trees are used to maintain the blocks efficiently and to keep track of empty space.

Hornet implements the bulk update scheme in which bulk updates and graph queries alternate. The bulk update exploits parallelism for efficient usage of the GPU resources.

5.7 GraphOne [135]

GraphOne [135] focuses on the parallel *efficient* execution of global graph algorithms (such as PageRank) and streaming graph updates. The graph is maintained using a variant of an adjacency list. To process graph updates, the updates are first stored in an edge list and also written to disk for persistence. If this edge list exceeds a certain storage threshold, the updates are applied as a batch in parallel to the adjacency list. Both the edge list and the adjacency list are stored in memory. The core idea is to exploit the fast edge list for immediate updates and stream processing, and provide snapshots of the adjacency list for long running graph analytics. Since multiple snapshots of the adjacency list can be created in a lightweight way, queries are processed immediately when they arrive.

5.8 Aspen [63]

The Aspen framework [63] uses a novel data structure called the *C-tree* to store graph structures. C-tree is based on a *purely-functional compressed search tree*. A functional search tree is a tree data structure that can be expressed only by mathematical functions, which makes the data structure immutable (since a mathematical function must always return the same result for the same input, independently of any state associated with the data structure). Furthermore, functional search trees offer lightweight snapshots, provably efficient running times, and they facilitate concurrent processing of queries and updates. Now, the C-tree extends purely-functional search trees: it overcomes the poor space usage and low locality. Elements represented by the tree are stored in *chunks* and each chunk is stored contiguously in an array, leading to improved locality. To improve the space usage, chunks can be compressed by applying difference encoding, since each block stores a sorted set of integers.

In Aspen, a graph is represented as a tree-of-trees: A purely-functional tree stores the set of vertices (vertex-tree) and each vertex stores the edges in its own C-tree (edge-tree). Additional information is stored in the vertex-tree such that basic graph structural properties, such as the total number of edges and vertices, can be queried in constant time. Similarly, the trees can be augmented to store properties (such as weights), but it is omitted in the described work. For algorithms that operate on the whole graph (such as BFS), it is possible to precompute a *flat snapshot*: instead of accessing all vertices by querying the vertex-tree, an array is used to directly store the pointers to the vertices.

5.9 Tegra [110]

Tegra [110] is a system that enables graph analysis based on graph updates that are a part of any window of time. The framework is implemented on top of Apache Spark [212] that handles scheduling and work distribution. The core data structure of Tegra is an adaptive radix tree - a tree data structure that enables efficient updates and range scans. It allows to map a graph efficiently by storing it in *two* trees (a vertex tree and an edge tree) and creating lightweight snapshots by generating a new root node that holds the differences. The graph is partitioned (by the hash of the vertex ID) among compute nodes. The API allows the user to create new snapshots of the graph. The system can also automatically create snapshots when a certain limit of changes is reached. Therefore, queries and updates (that can be ingested from main memory or graph databases) run concurrently. The framework also stores the changes that happened in-between such snapshots, allowing to restore any state and apply computations on any window. Since the snapshots take a lot of memory, they are written to disk using the last recently used policy.

5.10 Apache Flink [48]

Apache Flink [48] is a general purpose streaming system for streaming *and* batch computations (on both bounded and unbounded streams). These two concepts are usually considered different, but Flink treats them similarly. Two user APIs are available for implementation: the `DataSet` API for batch processing and the `DataStream` API for unbounded stream processing. A variety of custom operators can be implemented, allowing to maintain computation state, define iterative dataflows, compute over a stream window, and implement algorithms from the Bulk Synchronous Parallel model [201]. Both APIs generate programs that are represented as a directed acyclic graph of operators connected by data streams. Since operators can keep state and the system makes no assumption over the input streams, it is suited for graph streaming for rich data (edge and vertex properties), and it enables the user to update the graph and execute a broad range of graph algorithms.

5.11 Others

Other streaming frameworks come with similar design tradeoffs and features [74], [132], [206], [18], [214], [207], [149], [114], [180], [112], [102]. We now briefly describe examples, providing a starting point for further reading. **GraphInc** [46] is a framework built on top of Giraph [150]

that enables the developer to develop programs using the vertex-centric abstraction, which is then executed by the runtime over dynamic graphs. **UNICORN** [195] is a system that relies on InfoSphere, a large-scale, distributed data stream processing middleware developed at IBM Research. **DeltaGraph** [62] is a Haskell library for graph processing, which performs graph updates *lazily*. **iGraph** [115] is a system implemented on top of Apache Spark [212] and GraphX [91] that focuses on hash-based vertex-cut partitioning strategies for dynamic graphs, and proposes to use the vertex-centric programming model for such graphs. However, it is unclear on the details of developing different graph algorithms with the proposed approach. **EvoGraph** [183] is a simple extension of GraphIn. Whenever a batch of updates arrives, EvoGraph decides whether to use an incremental form of updating its structure, similar to that in GraphIn, or whether to recompute the maintained graph stored as an AM. **Sprouter** [3] is another system built on top of Spark. **PAST** [66] is a framework for processing *spatio-temporal graphs* with *up to 100 trillion edges* that track people, locations, and their connections. It relies on the underlying Cassandra storage [137].

5.12 Summary and Design Insights

In the majority of considered frameworks, the **representation** of the graph structure is usually some combination of the adjacency or the edge list that consists of *chunks* (often called *blocks*) with edges stored contiguously. 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. Moreover, most frameworks use some form of **batching** the updates to increase the parallelism and ultimately the throughput of graph accesses.

6 Relations to Graph Databases

We now discuss key differences between graph streaming frameworks and graph databases. We refer the reader to a recent survey on the latter class of systems [35], which provides details of native graph databases such as Neo4j [177], RDF stores [56], and other types of NoSQL stores used for managing graphs. We also exclude RDF streaming designs as we identify them to be strongly related to the domain of database systems, and point the reader to respective publications for more details [90], [43], [134], [47].

6.1 Graph Databases vs. Graph Streaming Systems

Graph databases usually deal with **complex and rich graph models** (such as the Labeled Property Graph [17] or Resource Description Framework [56]) 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. Second, graph databases often include **transactional support**, ensuring ACID properties [35], [99]. Third, despite a lack of agreement on a single language for querying graph databases, all the languages (e.g., SPARQL [172], Gremlin [178], Cypher [83], [106], and SQL [58]) provide rich support for **pattern matching queries** [72] or **business intelligence queries** [196]. Moreover, graph database systems maintain complex distributed **index structures** to accelerate different forms of queries.

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, data models are usually simple, without support for arbitrary attached properties. Second, there is no common support for any form of ACID, and the graph updates, even if sometimes they also referred to as transactions [216], are usually **lightweight**: single edge insertions or deletions, rather than arbitrary pattern matching queries common in graph database workloads. Moreover, contrarily to graph databases, streaming frameworks put more focus on **high velocity updates** that can be rapidly ingested into the graph structure. Finally, even if streaming frameworks often offer indexing, the supported index structures are simple [71].

6.2 Systems Combining Both Areas

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

Concerto [139] 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 [9], 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 [127] is a framework with focus on memory-efficient storage. It builds on Succint [6], 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** [216] 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 Specific Streaming Solutions

There are works on streaming and dynamic graphs that focus on solving a specific graph problem in a dynamic setting. Details of such solutions are outside the core focus of this survey. We outline them as a reference point for the reader. First, different designs target effective **partitioning of streaming graph datasets** [173], [168], [211], [79], [78], [77], [107], [189], [105], [152], [170]. Second, different works focus on solving a **specific graph problem in a streaming setting**. Targeted problems include graph clustering [103], mining periodic cliques [174], search for persistent communities [140], [176], tracking conductance [84], event pattern [166] and subgraph [162] discovery, solving ego-centric queries [161], pattern detection [53], [85], [186], [131], [141], [194], [54], [86], densest subgraph identification [113], frequent subgraph mining [19], dense subgraph detection [145], construction and querying of knowledge graphs [52], stream summarization [92], graph sparsification [11], [25], k-core maintenance [13], shortest paths [193], Betweenness Centrality [104], [199], [192], Triangle Counting [147], Katz Centrality [203], mincuts [133], [89] Connected Components [151], or PageRank [97], [55].

8 Challenges

Many research challenges related to streaming graph frameworks are similar to those in graph databases [35].

First, one should identify the most beneficial design choices for different use cases in the domain of streaming and dynamic graph processing. As shown in this paper, 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.

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, programming abstractions [197], and paradigms. 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 [26], [38], [59], high-performance networking hardware and associated abstractions [64], [31], [27], [181], [28], [87], low-cost atomics [165], [182], hardware transactions [30], and others [27], [10]. One could also investigate topology-aware or routing-aware data distribution for graph streaming, especially together with recent high-performance network topologies [29], [130] and routing [37], [144], [88]. Finally, ensuring speedups due to different data modeling abstractions, such as the algebraic abstraction [126], [33], [34], [136], may be a promising direction.

Finally, an interesting question is whether graph databases are inherently different from streaming frameworks. While merging these two classes of systems is an interesting research goal with many potential benefits, the difference in related 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. Is it 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.

Acknowledgements

We thank Khuzaima Daudjee for his suggestions regarding related work. We thank PRODYNA AG (Darko Križić, Jens Nixdorf, and Christoph Körner) for generous support.

References

[1] Apache Giraph Project. https://giraph.apache.org/.
[2] Z. Abbas, V. Kalavri, P. Carbone, and V. Vlassov. Streaming graph partitioning: an experimental study. *Proceedings of the VLDB Endowment*, 11(11):1590–1603, 2018.
[3] T. Abughofa and F. Zulkernine. Sprouter: Dynamic graph processing over data streams at scale. In *DEXA*, pages 321–328. Springer, 2018.

[4] U. A. Acar, D. Anderson, G. E. Blelloch, and L. Dhulipala. Parallel batch-dynamic graph connectivity. In *ACM SPAA*, pages 381–392, 2019.

[5] U. A. Acar, A. Cotter, B. Hudson, and D. Türkoglu. Parallelism in dynamic well-spaced point sets. In *ACM SPAA*, pages 33–42, 2011.

[6] R. Agarwal et al. Succinct: Enabling queries on compressed data. In *NSDI*, pages 337–350, 2015.

[7] C. Aggarwal and K. Subbian. Evolutionary network analysis: A survey. *ACM Computing Surveys (CSUR)*, 47(1):10, 2014.

[8] G. Aggarwal, M. Datar, S. Rajagopalan, and M. Ruhl. On the streaming model augmented with a sorting primitive. In *IEEE FOCS*, pages 540–549, 2004.

[9] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *ACM SIGOPS Op. Sys. Rev.*, 2007.

[10] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi. A scalable processing-in-memory accelerator for parallel graph processing. *ACM SIGARCH Comp. Arch. News*, 2016.

[11] K. J. Ahn and S. Guha. Graph sparsification in the semi-streaming model. In *ICALP*, pages 328–338. Springer, 2009.

[12] K. J. Ahn, S. Guha, and A. McGregor. Graph sketches: sparsification, spanners, and subgraphs. In *ACM PODS*, pages 5–14, 2012.

[13] H. Aksu, M. Canim, Y.-C. Chang, I. Korpeoglu, and Ö. Ulusoy. Distributed k-core view materialization and maintenance for large dynamic graphs. *IEEE TKDE*, 26(10):2439–2452, 2014.

[14] K. Ammar. Techniques and systems for large dynamic graphs. In *SIGMOD'16 PhD Symposium*, pages 7–11. ACM, 2016.

[15] K. Ammar, F. McSherry, S. Salihoglu, and M. Joglekar. Distributed evaluation of subgraph queries using worst-case optimal low-memory dataflows. *VLDB*, 11(6):691–704, 2018.

[16] A. Andoni, J. Chen, R. Krauthgamer, B. Qin, D. P. Woodruff, and Q. Zhang. On sketching quadratic forms. In *ACM ITCS*, pages 311–319, 2016.

[17] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. Reutter, and D. Vrgoč. Foundations of Modern Query Languages for Graph Databases. *in ACM Comput. Surv.*, 50(5):68:1–68:40, 2017.

[18] S. Aridhi et al. Bladyg: A graph processing framework for large dynamic graphs. *Big data research*, 9:9–17, 2017.

[19] C. Aslay, M. A. U. Nasir, G. De Francisci Morales, and A. Gionis. Mining frequent patterns in evolving graphs. In *ACM CIKM*, pages 923–932, 2018.

[20] S. Assadi, S. Khanna, and Y. Li. On estimating maximum matching size in graph streams. *SODA*, 2017.

[21] S. Assadi, S. Khanna, Y. Li, and G. Yaroslavtsev. Maximum matchings in dynamic graph streams and the simultaneous communication model. 2016.

[22] 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 Computing*, 18(3):1189–1213, 2015.

[23] S. Behnezhad, M. Derakhshan, M. Hajiaghayi, C. Stein, and M. Sudan. Fully dynamic maximal independent set with polylogarithmic update time. *FOCS*, 2019.

[24] 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. *arXiv preprint arXiv:1901.10183*, 2019.

[25] M. Besta et al. Slim graph: Practical lossy graph compression for approximate graph processing, storage, and analytics. 2019.

[26] M. Besta, M. Fischer, T. Ben-Nun, J. De Fine Licht, and T. Hoefler. Substream-centric maximum matchings on fpga. In *ACM/SIGDA FPGA*, pages 152–161, 2019.

[27] 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. In *ACM SIGPLAN Notices*, 2018.

[28] M. Besta and T. Hoefler. Fault tolerance for remote memory access programming models. In *ACM HPDC*, pages 37–48, 2014.

[29] M. Besta and T. Hoefler. Slim fly: A cost effective low-diameter network topology. In *ACM/IEEE Supercomputing*, pages 348–359, 2014.

[30] M. Besta and T. Hoefler. Accelerating irregular computations with hardware transactional memory and active messages. In *ACM HPDC*, 2015.

[31] M. Besta and T. Hoefler. Active access: A mechanism for high-performance distributed data-centric computations. In *ACM ICS*, 2015.

[32] M. Besta and T. Hoefler. Survey and taxonomy of lossless graph compression and space-efficient graph representations. *arXiv preprint arXiv:1806.01799*, 2018.

[33] M. Besta, R. Kanakagiri, H. Mustafa, M. Karasikov, G. Rätsch, T. Hoefler, and E. Solomonik. Communication-efficient jaccard similarity for high-performance distributed genome comparisons. *arXiv preprint arXiv:1911.04200*, 2019.

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

[35] M. Besta, E. Peter, R. Gerstenberger, M. Fischer, M. Podstawski, C. Barthels, G. Alonso, and T. Hoefler. Demystifying graph databases: Analysis and taxonomy of data organization, system designs, and graph queries. *arXiv preprint arXiv:1910.09017*, 2019.

[36] 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 *ACM HPDC*, 2017.

[37] M. Besta, M. Schneider, K. Cynk, M. Konieczny, E. Henriksson, S. Di Girolamo, A. Singla, and T. Hoefler. Fatpaths: Routing in supercomputers, data centers, and clouds with low-diameter networks when shortest paths fall short. *arXiv preprint arXiv:1906.10885*, 2019.

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

[39] M. Besta, D. Stanojevic, T. Zivic, J. Singh, M. Hoerold, and T. Hoefler. Log (graph): a near-optimal high-performance graph representation. In *PACT*, pages 7–1, 2018.

[40] S. Bhattacharya, M. Henzinger, and D. Nanongkai. A new deterministic algorithm for dynamic set cover. *FOCS*, 2019.

[41] S. Bhattacharya, M. Henzinger, D. Nanongkai, and C. Tsourakakis. Space-and time-efficient algorithm for maintaining dense subgraphs on one-pass dynamic streams. In *ACM STOC*, 2015.

[42] P. Boldi and S. Vigna. The webgraph framework i: compression techniques. In *ACM WWW*, pages 595–602, 2004.

[43] J. Broekstra et al. Sesame: A generic architecture for storing and querying rdf and rdf schema. In *ISWC*, pages 54–68. Springer, 2002.

[44] M. Bury, E. Grigorescu, A. McGregor, M. Monemizadeh, C. Schwiegelshohn, S. Vorotnikova, and S. Zhou. Structural results on matching estimation with applications to streaming. *Algorithmica*, 81(1):367–392, 2019.

[45] F. Busato, O. Green, N. Bombieri, and D. A. Bader. Hornet: An efficient data structure for dynamic sparse graphs and matrices on gpus. In *IEEE HPEC*, pages 1–7, 2018.

[46] Z. Cai, D. Logothetis, and G. Siganos. Facilitating real-time graph mining. In *ACM CloudDB*, pages 1–8, 2012.

[47] J.-P. Calbimonte, O. Corcho, and A. J. Gray. Enabling ontology-based access to streaming data sources. In *Springer ISWC*, 2010.

[48] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink: Stream and batch processing in a single engine. *IEEE-CS Bull. Tech. Com. on Data Eng.*, 2015.

[49] S. Chechik and T. Zhang. Fully dynamic maximal independent set in expected poly-log update time. *FOCS*, 2019.

[50] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen. Kineograph: taking the pulse of a fast-changing and connected world. In *ACM EuroSys*, pages 85–98, 2012.

[51] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *Proceedings of the VLDB Endowment*, 8(12):1804–1815, 2015.

[52] S. Choudhury, K. Agarwal, S. Purohit, B. Zhang, M. Pirrung, W. Smith, and M. Thomas. Nous: Construction and querying of dynamic knowledge graphs. In *IEEE ICDE*, pages 1563–1565, 2017.

[53] S. Choudhury, L. Holder, G. Chin, K. Agarwal, and J. Feo. A selectivity based approach to continuous pattern detection in streaming graphs. *arXiv preprint arXiv:1503.00849*, 2015.

[54] S. Choudhury, S. Purohit, P. Lin, Y. Wu, L. Holder, and K. Agarwal. Percolator: Scalable pattern discovery in dynamic graphs. In *ACM WSDM*, pages 759–762, 2018.

[55] M. E. Coimbra, R. Rosa, S. Esteves, A. P. Francisco, and L. Veiga. Graphbolt: Streaming graph approximations on big data. *arXiv preprint arXiv:1810.02781*, 2018.

[56] R. Cyganiak, D. Wood, and M. Lanthaler. RDF 1.1 Concepts and Abstract Syntax. Available at https://www.w3.org/TR/rdf11-concepts/.

[57] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. *SIAM journal on computing*, 31(6):1794–1813, 2002.

[58] C. J. Date and H. Darwen. *A Guide to the SQL Standard*, volume 3. Addison-Wesley New York, 1987.

[59] J. de Fine Licht et al. Transformations of high-level synthesis codes for high-performance computing. *arXiv:1805.08288*, 2018.

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

[61] C. Demetrescu, I. Finocchi, and A. Ribichini. Trading off space for passes in graph streaming problems. *ACM TALG*, 6(1):6, 2009.

[62] P. Dexter, Y. D. Liu, and K. Chiu. Lazy graph processing in haskell. In *ACM SIGPLAN Notices*, volume 51, pages 182–192. ACM, 2016.

[63] L. Dhulipala et al. Low-latency graph streaming using compressed purely-functional trees. *arXiv:1904.08380*, 2019.

[64] S. Di Girolamo, K. Taranov, A. Kurth, M. Schaffner, T. Schneider, J. Beránek, M. Besta, L. Benini, D. Roweth, and T. Hoefler. Network-accelerated non-contiguous memory transfers. *arXiv preprint arXiv:1908.08590*, 2019.

[65] L. Di Paola, M. De Ruvo, P. Paci, D. Santoni, and A. Giuliani. Protein contact networks: an emerging paradigm in chemistry. *Chemical reviews*, 113(3):1598–1613, 2012.

[66] M. Ding et al. Storing and querying large-scale spatio-temporal graphs with high-throughput edge insertions. *arXiv:1904.09610*, 2019.

[67] N. Doekemeijer and A. L. Varbanescu. A survey of parallel graph processing frameworks. *Delft University of Technology*, page 21, 2014.

[68] R. Duan, H. He, and T. Zhang. Dynamic edge coloring with improved approximation. In *ACM-SIAM SODA*, pages 1937–1945, 2019.

[69] D. Durfee, L. Dhulipala, J. Kulkarni, R. Peng, S. Sawlani, and X. Sun. Parallel batch-dynamic graphs: Algorithms and lower bounds. *SODA*, 2020.

[70] D. Durfee, Y. Gao, G. Goranci, and R. Peng. Fully dynamic spectral vertex sparsifiers and applications. In *ACM STOC*, pages 914–925, 2019.

[71] D. Ediger, R. McColl, J. Riedy, and D. A. Bader. Stinger: High performance data structure for streaming graphs. In *IEEE HPEC*, pages 1–5, 2012.

[72] O. Erling, A. Averbuch, J. Larriba-Pey, H. Chafi, A. Gubichev, A. Prat, M.-D. Pham, and P. Boncz. The LDBC Social Network Benchmark: Interactive Workload. in SIGMOD, pages 619–630, 2015.

[73] 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*, 2018.

[74] J. Fairbanks, D. Ediger, R. McColl, D. A. Bader, and E. Gilbert. A statistical framework for streaming graph analysis. In *IEEE/ACM ASONAM*, pages 341–347, 2013.

[75] J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang. On graph problems in a semi-streaming model. *Theoretical Computer Science*, 348(2-3):207–216, 2005.

[76] G. Feng et al. Distinger: A distributed graph data structure for massive dynamic graph processing. In *IEEE Big Data*, pages 1814–1822, 2015.

[77] I. Filippidou and Y. Kotidis. Online and on-demand partitioning of streaming graphs. In *IEEE Big Data*, pages 4–13.

[78] H. Firth and P. Missier. Workload-aware streaming graph partitioning. In *EDBT/ICDT Workshops*. Citeseer, 2016.

[79] H. Firth, P. Missier, and J. Aiston. Loom: Query-aware partitioning of online graphs. *arXiv preprint arXiv:1711.06608*, 2017.

[80] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. In *Discrete Mathematics and Theoretical Computer Science*, pages 137–156, 2007.

[81] S. Forster and G. Goranci. Dynamic low-stretch trees via dynamic low-diameter decompositions. In *ACM STOC*, pages 377–388, 2019.

[82] F. Fouquet, T. Hartmann, S. Mosser, and M. Cordy. Enabling lock-free concurrent workers over temporal graphs composed of multiple time-series. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, pages 1054–1061. ACM, 2018.

[83] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor. Cypher: An evolving query language for property graphs. In *ACM SIGMOD*, pages 1433–1445, 2018.

[84] S. Galhotra, A. Bagchi, S. Bedathur, M. Ramanath, and V. Jain. Tracking the conductance of rapidly evolving topic-subgraphs. *Proc. VLDB*, 8(13):2170–2181, 2015.

[85] J. Gao, C. Zhou, and J. X. Yu. Toward continuous pattern detection over evolving large graph with snapshot isolation. *The VLDB Journal—The International Journal on Very Large Data Bases*, 25(2):269–290, 2016.

[86] J. Gao, C. Zhou, J. Zhou, and J. X. Yu. Continuous pattern detection over billion-edge graph using distributed framework. In *IEEE ICDE*, pages 556–567, 2014.

[87] R. Gerstenberger, M. Besta, and T. Hoefler. Enabling Highly-scalable Remote Memory Access Programming with MPI-3 One Sided. In *ACM/IEEE Supercomputing*, SC '13, pages 53:1–53:12, 2013.

[88] S. Ghorbani, Z. Yang, P. Godfrey, Y. Ganjali, and A. Firoozshahian. Drill: Micro load balancing for low-latency data center networks. In *ACM SIGCOMM*, pages 225–238, 2017.

[89] L. Gianinazzi, P. Kalvoda, A. De Palma, M. Besta, and T. Hoefler. Communication-avoiding parallel minimum cuts and connected components. In *ACM SIGPLAN Notices*, volume 53, pages 219–232. ACM, 2018.

[90] F. Goasdoué, I. Manolescu, and A. Roatiş. Efficient query answering against dynamic rdf databases. In *ACM EDBT*, pages 299–310, 2013.

[91] 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 *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 599–613, 2014.

[92] X. Gou, L. Zou, C. Zhao, and T. Yang. Fast and accurate graph stream summarization. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1118–1129. IEEE, 2019.

[93] O. Green and D. A. Bader. custinger: Supporting dynamic graph algorithms for gpus. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2016.

[94] W. Gropp, T. Hoefler, T. Rajeev, and E. Lusk. *Using Advanced MPI: Modern Features of the Message-Passing Interface*. The MIT Press, 2014.

[95] S. Guha and A. McGregor. Graph synopses, sketches, and streams: A survey. *PVLDB*, 5(12):2030–2031, 2012.

[96] S. Guha, A. McGregor, and D. Tench. Vertex and hyperedge connectivity in dynamic graph streams. In *ACM PODS*, pages 241–247, 2015.

[97] W. Guo, Y. Li, M. Sha, and K.-L. Tan. Parallel personalized pagerank on dynamic graphs. *Proceedings of the VLDB Endowment*, 11(1):93–106, 2017.

[98] M. Han and K. Daudjee. Giraph unchained: barrierless asynchronous parallel execution in pregel-like graph processing systems. *Proceedings of the VLDB Endowment*, 8(9):950–961, 2015.

[99] M. Han and K. Daudjee. Providing serializability for pregel-like graph processing systems. In *EDBT*, pages 77–88, 2016.

[100] M. Han, K. Daudjee, K. Ammar, M. T. Özsu, X. Wang, and T. Jin. An experimental comparison of pregel-like graph processing systems. *Proc. VLDB*, 7(12):1047–1058, 2014.

[101] W. Han, Y. Miao, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, W. Chen, and E. Chen. Chronos: a graph engine for temporal graph analysis. In *9th European Conference on Computer Systems*, page 1. ACM, 2014.

[102] T. Hartmann, F. Fouquet, M. Jimenez, R. Rouvoy, and Y. Le Traon. Analyzing complex data in motion at scale with temporal graphs. 2017.

[103] M. Hassani, P. Spaus, A. Cuzzocrea, and T. Seidl. I-hastream: density-based hierarchical clustering of big data streams and its application to big graph analytics tools. In *CCGrid*, pages 656–665. IEEE, 2016.

[104] T. Hayashi, T. Akiba, and Y. Yoshida. Fully dynamic betweenness centrality maintenance on massive networks. *Proceedings of the VLDB Endowment*, 9(2):48–59, 2015.

[105] L. Hoang, R. Dathathri, G. Gill, and K. Pingali. Cusp: A customizable streaming edge partitioner for distributed graph analytics. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 439–450. IEEE, 2019.

[106] F. Holzschuher and R. Peinl. Performance of graph query languages: comparison of cypher, gremlin and native access in neo4j. In *Proceedings of the Joint EDBT/ICDT 2013 Workshops*, pages 195–204. ACM, 2013.

[107] J. Huang and D. J. Abadi. Leopard: Lightweight edge-oriented partitioning and replication for dynamic graphs. *Proceedings of the VLDB Endowment*, 9(7):540–551, 2016.

[108] G. F. Italiano, S. Lattanzi, V. S. Mirrokni, and N. Parotsidis. Dynamic algorithms for the massively parallel computation model. In *The 31st ACM on Symposium on Parallelism in Algorithms and Architectures*, pages 49–58. ACM, 2019.

[109] A. Iyer, L. E. Li, and I. Stoica. Celliq: Real-time cellular network analytics at scale. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, pages 309–322, 2015.

[110] A. Iyer, Q. Pu, K. Patel, J. Gonzalez, and I. Stoica. Tegra: Efficient ad-hoc analytics on time-evolving graphs. Technical report, Technical report, 2019.

[111] A. P. Iyer, L. E. Li, T. Das, and I. Stoica. Time-evolving graph processing at scale. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, page 5. ACM, 2016.

[112] S. Ji et al. A low-latency computing framework for time-evolving graphs. *The Journal of Supercomputing*, 75(7):3673–3692, 2019.

[113] H. Jin, C. Lin, H. Chen, and J. Liu. Quickpoint: Efficiently identifying densest sub-graphs in online social networks for event stream dissemination. *IEEE Transactions on Knowledge and Data Engineering*, 2018.

[114] P. Joaquim. Hourglass-incremental graph processing on heterogeneous infrastructures.

[115] W. Ju, J. Li, W. Yu, and R. Zhang. igraph: an incremental data processing system for dynamic graph. *Frontiers of Computer Science*, 10(3):462–476, 2016.

[116] V. Kalavri, V. Vlassov, and S. Haridi. High-level programming abstractions for distributed graph processing. *IEEE Transactions on Knowledge and Data Engineering*, 30(2):305–324, 2017.

[117] J. Kallaugher, M. Kapralov, and E. Price. The sketching complexity of graph and hypergraph counting. *FOCS*, 2018.

[118] S. Kamburugamuve and G. Fox. Survey of distributed stream processing. *Bloomington: Indiana University*, 2016.

[119] D. M. Kane, K. Mehlhorn, T. Sauerwald, and H. Sun. Counting arbitrary subgraphs in data streams. In *International Colloquium on Automata, Languages, and Programming*, pages 598–609. Springer, 2012.

[120] M. Kapralov, S. Khanna, and M. Sudan. Approximating matching size from random streams. *SODA*, 2014.

[121] M. Kapralov, S. Mitrovic, A. Norouzi-Fard, and J. Tardos. Space efficient approximation to maximum matching size from uniform edge samples. *SODA*, 2020.

[122] M. Kapralov, A. Mousavifar, C. Musco, C. Musco, N. Nouri, A. Sidford, and J. Tardos. Fast and space efficient spectral sparsification in dynamic streams. *SODA*, abs/1903.12150, 2020.

[123] M. Kapralov, N. Nouri, A. Sidford, and J. Tardos. Dynamic streaming spectral sparsification in nearly linear time and space. *CoRR*, abs/1903.12150, 2019.

[124] M. Kapralov and D. P. Woodruff. Spanners and sparsifiers in dynamic streams. *PODC*, 2014.

[125] H. Karloff, S. Suri, and S. Vassilvitskii. A model of computation for mapreduce. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*, pages 938–948. SIAM, 2010.

[126] 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 *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–9. IEEE, 2016.

[127] A. Khandelwal, Z. Yang, E. Ye, R. Agarwal, and I. Stoica. Zipg: A memory-efficient graph store for interactive queries. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1149–1164. ACM, 2017.

[128] U. Khurana and A. Deshpande. Efficient snapshot retrieval over historical graph data. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 997–1008. IEEE, 2013.

[129] U. Khurana and A. Deshpande. Storing and analyzing historical graph data at scale. *arXiv preprint arXiv:1509.08960*, 2015.

[130] J. Kim, W. J. Dally, S. Scott, and D. Abts. Technology-driven, highly-scalable dragonfly topology. In *2008 International Symposium on Computer Architecture*, pages 77–88. IEEE, 2008.

[131] K. Kim, I. Seo, W.-S. Han, J.-H. Lee, S. Hong, H. Chafi, H. Shin, and G. Jeong. Turboflux: A fast continuous subgraph matching system for streaming graph data. In *Proceedings of the 2018 International Conference on Management of Data*, pages 411–426. ACM, 2018.

[132] J. King, T. Gilray, R. M. Kirby, and M. Might. Dynamic sparse-matrix allocation on gpus. In *International Conference on High Performance Computing*, pages 61–80. Springer, 2016.

[133] D. Kogan and R. Krauthgamer. Sketching cuts in graphs and hypergraphs. In *Proceedings of the 2015 Conference on Innovations in Theoretical Computer Science*, pages 367–376. ACM, 2015.

[134] S. Komazec, D. Cerri, and D. Fensel. Sparkwave: continuous schema-enhanced pattern matching over rdf data streams. In *6th ACM International Conference on Distributed Event-Based Systems*, pages 58–68. ACM, 2012.

[135] P. Kumar and H. H. Huang. Graphone: A data store for real-time analytics on evolving graphs. In *17th {USENIX} Conference on File and Storage Technologies ({FAST} 19)*, pages 249–263, 2019.

[136] G. Kwasniewski, M. Kabić, M. Besta, J. VandeVondele, R. Solcà, and T. Hoefler. Red-blue pebbling revisited: near optimal parallel matrix-matrix multiplication. In *ACM/IEEE Supercomputing*, page 24. ACM, 2019.

[137] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[138] K. G. Larsen, J. Nelson, H. L. Nguyen, and M. Thorup. Heavy hitters via cluster-preserving clustering. *Commun. ACM*, 62(8):95–100, 2019.

[139] M. M. Lee, I. Roy, A. AuYoung, V. Talwar, K. Jayaram, and Y. Zhou. Views and transactional storage for large graphs. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 287–306. Springer, 2013.

[140] R.-H. Li, J. Su, L. Qin, J. X. Yu, and Q. Dai. Persistent community search in temporal networks. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 797–808. IEEE, 2018.

[141] Y. Li, L. Zou, M. T. Özsu, and D. Zhao. Time constrained continuous subgraph search over streaming graphs. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1082–1093. IEEE, 2019.

[142] 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 *ACM/IEEE Supercomputing*, page 56. IEEE Press, 2018.

[143] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. E. Guestrin, and J. Hellerstein. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1408.2041*, 2014.

[144] 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 *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, pages 357–371, 2018.

[145] S. Ma, R. Hu, L. Wang, X. Lin, and J. Huai. Fast computation of dense temporal subgraphs. In *ICDE*, pages 361–372. IEEE, 2017.

[146] P. Macko, V. J. Marathe, D. W. Margo, and M. I. Seltzer. Llama: Efficient graph analytics using large multiversioned arrays. In *2015 IEEE 31st International Conference on Data Engineering*, pages 363–374. IEEE, 2015.

[147] D. Makkar, D. A. Bader, and O. Green. Exact and parallel triangle counting in dynamic graphs. In *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*, pages 2–12. IEEE, 2017.

[148] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[149] M. Mariappan and K. Vora. Graphbolt: Dependency-driven syn-

chronous processing of streaming graphs. In *Proceedings of the Fourteenth EuroSys Conference 2019*, page 25. ACM, 2019.

[150] C. Martella, R. Shaposhnik, D. Logothetis, and S. Harenberg. *Practical graph analytics with apache giraph*, volume 1. Springer, 2015.

[151] R. McColl, O. Green, and D. A. Bader. A new parallel algorithm for connected components in dynamic graphs. In *20th Annual International Conference on High Performance Computing*, pages 246–255. IEEE, 2013.

[152] A. McCrabb, E. Winsor, and V. Bertacco. Dredge: Dynamic repartitioning during dynamic graph execution. In *Proceedings of the 56th Annual Design Automation Conference 2019*, page 28. ACM, 2019.

[153] R. R. McCune, T. Weninger, and G. Madey. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys (CSUR)*, 48(2):25, 2015.

[154] A. McGregor. Graph stream algorithms: a survey. *ACM SIGMOD Record*, 43(1):9–20, 2014.

[155] A. McGregor. Graph stream algorithms: a survey. *SIGMOD Record*, 43(1):9–20, 2014.

[156] F. McSherry, D. G. Murray, R. Isaacs, and M. Isard. Differential dataflow. In *CIDR*, 2013.

[157] Y. Miao, W. Han, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, E. Chen, and W. Chen. Immortalgraph: A system for storage and analysis of temporal graphs. *ACM Transactions on Storage (TOS)*, 11(3):14, 2015.

[158] O. Michail. An introduction to temporal graphs: An algorithmic perspective. *Internet Mathematics*, 12(4):239–280, 2016.

[159] G. T. Minton and E. Price. Improved concentration bounds for count-sketch. In *Proceedings of the twenty-fifth annual ACM-SIAM symposium on Discrete algorithms*, pages 669–686. Society for Industrial and Applied Mathematics, 2014.

[160] J. Mondal and A. Deshpande. Managing large dynamic graphs efficiently. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 145–156. ACM, 2012.

[161] J. Mondal and A. Deshpande. Eagr: Supporting continuous ego-centric aggregate queries over large dynamic graphs. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of data*, pages 1335–1346. ACM, 2014.

[162] J. Mondal and A. Deshpande. Casqd: continuous detection of activity-based subgraph pattern queries on dynamic graphs. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, pages 226–237. ACM, 2016.

[163] D. G. Murray, F. McSherry, M. Isard, R. Isaacs, P. Barham, and M. Abadi. Incremental, iterative data processing with timely dataflow. *Communications of the ACM*, 59(10):75–83, 2016.

[164] S. Muthukrishnan. Data streams: Algorithms and applications. *Foundations and Trends in Theoretical Computer Science*, 1(2), 2005.

[165] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim. Graphpim: Enabling instruction-level pim offloading in graph computing frameworks. In *2017 IEEE International symposium on high performance computer architecture (HPCA)*, pages 457–468. IEEE, 2017.

[166] M. H. Namaki, P. Lin, and Y. Wu. Event pattern discovery by keywords in graph streams. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 982–987. IEEE, 2017.

[167] S. Neuendorffer and K. Vissers. Streaming systems in FPGAs. In *Intl. Workshop on Embedded Computer Systems*, pages 147–156. Springer, 2008.

[168] D. Nicoara, S. Kamali, K. Daudjee, and L. Chen. Hermes: Dynamic partitioning for distributed social network graph databases. In *EDBT*, pages 25–36, 2015.

[169] T. C. O'connell. A survey of graph algorithms under extended streaming models of computation. In *Fundamental Problems in Computing*, pages 455–476. Springer, 2009.

[170] A. Pacaci and M. T. Özsu. Experimental analysis of streaming algorithms for graph partitioning. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1375–1392. ACM, 2019.

[171] P. Peng and C. Sohler. Estimating graph parameters from random order streams. 2018.

[172] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of sparql. *ACM TODS*, 34(3):16, 2009.

[173] F. Petroni, L. Querzoni, K. Daudjee, S. Kamali, and G. Iacoboni. HDRF: Stream-based partitioning for power-law graphs. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, pages 243–252. ACM, 2015.

[174] H. Qin, R.-H. Li, G. Wang, L. Qin, Y. Cheng, and Y. Yuan. Mining periodic cliques in temporal networks. In *ICDE*, pages 1130–1141. IEEE, 2019.

[175] C. Ren, E. Lo, B. Kao, X. Zhu, and R. Cheng. On querying historical evolving graph sequences. *Proceedings of the VLDB Endowment*, 4(11):726–737, 2011.

[176] J. Riedy and D. A. Bader. Multithreaded community monitoring for massive streaming graph data. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, pages 1646–1655. IEEE, 2013.

[177] I. Robinson, J. Webber, and E. Eifrem. Graph database internals. In *Graph Databases, Second Edition*, chapter 7, pages 149–170. O'Relly, 2015.

[178] M. A. Rodriguez. The gremlin graph traversal machine and language (invited talk). In *Proceedings of the 15th Symposium on Database Programming Languages*, pages 1–10. ACM, 2015.

[179] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 472–488. ACM, 2013.

[180] S. Sallinen, R. Pearce, and M. Ripeanu. Incremental graph processing for on-line analytics. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1007–1018. IEEE, 2019.

[181] P. Schmid, M. Besta, and T. Hoefler. High-performance distributed rma locks. In *ACM HPDC*, pages 19–30, 2016.

[182] H. Schweizer, M. Besta, and T. Hoefler. Evaluating the cost of atomic operations on modern architectures. In *IEEE PACT*, pages 445–456, 2015.

[183] D. Sengupta and S. L. Song. Evograph: On-the-fly efficient mining of evolving graphs on gpu. In *International Supercomputing Conference*, pages 97–119. Springer, 2017.

[184] D. Sengupta, N. Sundaram, X. Zhu, T. L. Willke, J. Young, M. Wolf, and K. Schwan. Graphin: An online high performance incremental graph processing framework. In *European Conference on Parallel Processing*, pages 319–333. Springer, 2016.

[185] M. Sha, Y. Li, B. He, and K.-L. Tan. Accelerating dynamic graph analytics on gpus. *Proceedings of the VLDB Endowment*, 11(1):107–120, 2017.

[186] M. Shao, J. Li, F. Chen, and X. Chen. An efficient framework for detecting evolving anomalous subgraphs in dynamic networks. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pages 2258–2266. IEEE, 2018.

[187] X. Shi, B. Cui, Y. Shao, and Y. Tong. Tornado: A system for real-time iterative analysis over evolving data. In *Proceedings of the 2016 International Conference on Management of Data*, pages 417–430. ACM, 2016.

[188] X. Shi, Z. Zheng, Y. Zhou, H. Jin, L. He, B. Liu, and Q.-S. Hua. Graph processing on gpus: A survey. *ACM Computing Surveys (CSUR)*, 50(6):81, 2018.

[189] Z. Shi, J. Li, P. Guo, S. Li, D. Feng, and Y. Su. Partitioning dynamic graph asynchronously with distributed fennel. *Future Generation Computer Systems*, 71:32–42, 2017.

[190] J. Shun and G. E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *ACM Sigplan Notices*, volume 48, pages 135–146. ACM, 2013.

[191] N. Simsiri, K. Tangwongsan, S. Tirthapura, and K. Wu. Work-efficient parallel union-find with applications to incremental graph connectivity. In *Euro-Par*, pages 561–573, 2016.

[192] E. Solomonik, M. Besta, F. Vella, and T. Hoefler. Scaling betweenness centrality using communication-efficient sparse matrix multiplication. In *ACM/IEEE Supercomputing*, page 47, 2017.

[193] S. Srinivasan, S. Riazi, B. Norris, S. K. Das, and S. Bhowmick. A shared-memory parallel algorithm for updating single-source shortest paths in large dynamic networks. In *HiPC*, pages 245–254. IEEE, 2018.

[194] X. Sun, Y. Tan, Q. Wu, and J. Wang. A join-cache tree based approach for continuous temporal pattern detection in streaming graph. In *ICSPCC*, pages 1–6. IEEE, 2017.

[195] T. Suzumura, S. Nishii, and M. Ganse. Towards large-scale graph stream processing platform. In *ACM WWW*, pages 1321–1326, 2014.

[196] G. Szárnyas, A. Prat-Pérez, A. Averbuch, J. Marton, M. Paradies, M. Kaufmann, O. Erling, P. Boncz, V. Haprian, and J. B. Antal. An Early Look at the LDBC Social Network Benchmark's Business Intelligence Workload. GRADES-NDA, pages 9:1–9:11, 2018.

[197] A. Tate et al. Programming abstractions for data locality. PADAL Workshop 2014, 2014.

[198] M. Then, T. Kersten, S. Günnemann, A. Kemper, and T. Neumann. Automatic algorithm transformation for efficient multi-snapshot analytics on temporal graphs. *Proceedings of the VLDB Endowment*, 10(8):877–888, 2017.

[199] A. Tripathy and O. Green. Scaling betweenness centrality in dynamic graphs. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2018.

[200] T. Tseng, L. Dhulipala, and G. Blelloch. Batch-parallel euler tour trees. In *2019 Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 92–106. SIAM, 2019.

[201] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[202] J. van den Brand and D. Nanongkai. Dynamic approximate shortest paths and beyond: Subquadratic and worst-case update time. *FOCS*, 2019.

[203] A. van der Grinten, E. Bergamini, O. Green, D. A. Bader, and H. Meyerhenke. Scalable katz ranking computation in large static and dynamic graphs. *arXiv preprint arXiv:1807.03847*, 2018.

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

[205] K. Vora, R. Gupta, and G. Xu. Synergistic analysis of evolving graphs. *ACM TACO*, 13(4):32, 2016.

[206] K. Vora, R. Gupta, and G. Xu. Kickstarter: Fast and accurate computations on streaming graphs via trimmed approximations. *ACM SIGOPS Operating Systems Review*, 51(2):237–251, 2017.

[207] 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 *USENIX OSDI*, pages 763–782, 2018.

[208] 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 *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, page 60. IEEE Press, 2018.

[209] M. Winter, R. Zayer, and M. Steinberger. Autonomous, independent management of dynamic graphs on gpus. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2017.

[210] W. Xie, Y. Tian, Y. Sismanis, A. Balmin, and P. J. Haas. Dynamic interaction graphs with probabilistic edge decay. In *IEEE ICDE*, pages 1143–1154, 2015.

[211] S. Yang, X. Yan, B. Zong, and A. Khan. Towards effective partition management for large graphs. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 517–528. ACM, 2012.

[212] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.

[213] J. Zhang. A survey on streaming algorithms for massive graphs. *Managing and Mining Graph Data*, pages 393–420, 2010.

[214] Y. Zhang, R. Chen, and H. Chen. Sub-millisecond stateful stream querying over fast-evolving linked data. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 614–630. ACM, 2017.

[215] S. Zhou, R. Kannan, H. Zeng, and V. K. Prasanna. An fpga framework for edge-centric graph processing. In *Proceedings of the 15th ACM International Conference on Computing Frontiers*, pages 69–77. ACM, 2018.

[216] 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. *arXiv preprint arXiv:1910.05773*, 2019.