# GraphOne: A Data Store for Real-time Analytics on Evolving Graphs

Pradeep Kumar and H. Howie Huang, *George Washington University*

https://www.usenix.org/conference/fast19/presentation/kumar

This paper is included in the Proceedings of the
17th USENIX Conference on File and Storage Technologies (FAST '19).
February 25–28, 2019 • Boston, MA, USA
978-1-939133-09-0

Open access to the Proceedings of the
17th USENIX Conference on File and
Storage Technologies (FAST '19)
is sponsored by

# GraphOne: A Data Store for Real-time Analytics on Evolving Graphs

Pradeep Kumar    H. Howie Huang
The George Washington University

## Abstract

There is a growing need to perform real-time analytics on evolving graphs in order to deliver the values of big data to users. The key requirement from such applications is to have a data store to support their diverse data access efficiently, while concurrently ingesting fine-grained updates at a high velocity. Unfortunately, current graph systems, either graph databases or analytics engines, are not designed to achieve high performance for both operations. To address this challenge, we have designed and developed GraphOne, a graph data store that combines two complementary graph storage formats (edge list and adjacency list), and uses *dual versioning* to decouple graph computations from updates. Importantly, it presents a new data abstraction, *GraphView*, to enable data access at two different granularities with only a small data duplication. Experimental results show that GraphOne achieves an ingestion rate of two to three orders of magnitude higher than graph databases, while delivering algorithmic performance comparable to a static graph system. GraphOne is able to deliver 5.36× higher update rate and over 3× better analytics performance compared to a state-of-the-art dynamic graph system.

## 1 Introduction

We live in a world where information networks have become an indivisible part of our daily lives. A large body of research has studied the relationships in such networks, e.g., biological networks [33], social networks [20, 41, 46], and web [9, 31]. In these applications, graph queries and analytics are being used to gain valuable insights from the data, which can be classified into two broad categories: *batch analytics* (e.g. PageRank [61], graph traversal [11, 49, 51]) that analyzes a static snapshot of the data, and *stream analytics* (e. g. anomaly detection [8], topic detection [64]) that studies the incoming data over a time window of interest. Generally speaking, batch analytics prefers a *base (data) store* that can provide indexed access on the non-temporal property of the graph such as the source vertex of an edge, and on the other hand, stream analytics needs a *stream (data) store* where data can be stored quickly and can be indexed by their arrival order for temporal analysis.

Increasingly, one needs to perform batch and stream processing together on evolving graphs [78, 68, 10, 69]. The key requirement here is to sustain a large volume of fine-grained updates at a high velocity, and simultaneously provide high-performance real-time analytics and query support.

This trend poses a number of challenges to the underlying storage and data management system. First, batch and stream analytics perform different kinds of data access, that is, the former visits the whole graph while the latter focuses on the data within a time window. Second, each analytic has a different notion of real time, that is, data is visible to the analytics at different granularity of data ingestion (updates). For example, an iterative algorithm such as PageRank can run on a graph that is updated at a coarse granularity, but a graph query to output the latest shortest path requires data visibility at a much finer granularity. Third, such a system should also be able to handle a high arrival rate of updates, and maintain data consistency while running concurrent batch and stream processing tasks.

Unfortunately, current graph systems can neither provide diverse data access nor at the right granularity in the presence of a high data arrival rate. Many dynamic graph systems [47, 54] only support batched updates, and a few others [21, 70] offer data visibility at fine granularity of updates but with a weak consistency guarantee, which as a result may cause an analytic iteration to run on different data versions and produce undesired results. Relational and graph databases such as Neo4j [59] can handle fine-grained updates, but suffer from poor ingestion rate for the sake of strong consistency guarantee [56]. Also, such systems are not designed to support high-performance streaming data access over a time window. On the other hand, graph stream engines [58, 17, 32, 72, 75, 67] interleave incremental computation with data ingestion, i.e., graph updates are batched and not applied until the end of an iteration. In short, the existing systems manage a private data store in a way to favor their specialized analytics.

In principle, one can utilize these specialized graph systems side-by-side to provide data management functions for dynamic graphs and support a wide spectrum of analytics and queries. However, such an approach would be suboptimal [78], as it is only as good as the weakest component, in many cases the graph database with poor performance for streaming data. Worse, this approach could also lead to excessive data duplication, as each subsystem would store a replica of the same underlying data in their own format.

In this work, we have designed GraphOne, a unified graph data store offering diverse data access at various granularity levels while supporting data ingestion at a high ar-

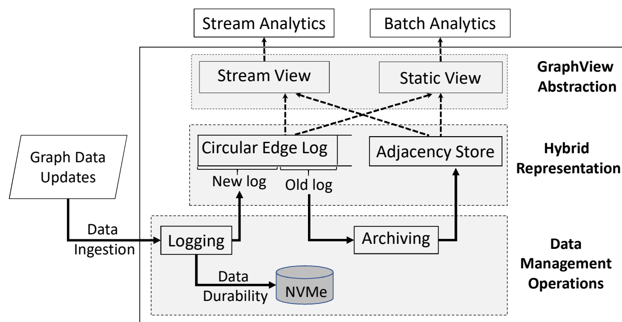


**Fig. 1:** High-level architecture of GRAPHONE. Solid and dotted arrows show the data management and access flow respectively.

rival rate. Fig. 1 provides a high-level overview. It leverages a *hybrid graph store* to combine a small circular edge log (henceforth *edge log*) and an *adjacency store* for their complementary advantages. Specifically, the edge log keeps the latest updates in the edge list format, and is designed to accelerate data ingestion. At the same time, the adjacency store holds the snapshots of the older data in the adjacency list format that is moved periodically from the edge log, and is optimized for batch and streaming analytics. It is important to note that the graph data is not duplicated in two formats, although a small amount of overlapping is allowed to keep the original composition of the versions intact.

GRAPHONE enforces data ordering using the temporal nature of the edge log, and keeps the per-vertex edge arrival order intact in the adjacency store. A *dual versioning* technique then exploits the fine-grained versioning of the edge list format and the coarse-grained versioning of the adjacency list format to create real-time versions. Further, GRAPHONE allows independent execution of analytics that run parallel to data management, and can fetch a new version at the end of its own incremental computation step. Additionally, we provide two optimization techniques, cacheline sized memory allocation and special handling of high degree vertices of power-law graphs, to reduce the memory requirement of versioned adjacency store.

GRAPHONE simplifies the diverse data access by presenting a new data abstraction, *GraphView*, on top of the hybrid store. Two types of GraphView are supported as shown in Fig. 1 : (1) the *static view* offers real-time versioning of the latest data for batch analytics; and (2) the *stream view* supports stream analytics with the most recent updates. These views offers *visibility* of data updates to analytics at two levels of granularity where the edge log is used to offer it at the edge level, while the adjacency store provides the same at coarse granularity of updates. As a result, GRAPHONE provides high-level applications with the flexibility to trade-off the granularity of data visibility for a desired performance. In other words, the edge log can be accessed if fine-grained data visibility is required, which can be tuned (§7.3).

We have implemented GRAPHONE as an in-memory graph datastore with a durability guarantee on external non-volatile memory express solid-state drives (NVMe SSD). For comparison, we have evaluated it against three types of in-memory graph systems: Neo4j and SQLite, two graph data management systems; Stinger [21], a dynamic graph system; and Galois [60], a static graph system, as well as GRAPHONE itself working with static graphs. The experimental results show that GRAPHONE can support a high data ingestion rate, specifically it achieves two to three orders of magnitude higher ingestion rate than graph databases, and 5.36× higher ingestion rate than Stinger. In addition, GRAPHONE outperforms Stinger by more than 3× on different analytics, and delivers equivalent algorithmic performance compared to Galois. The stream processing in GRAPHONE runs parallel to data ingestion which offers 26.22% higher ingestion rate compared to the current practice of interleaving the two.

To summarize, GRAPHONE makes three contributions:

- Unifies stream and base stores to manage the graph data in a dynamic environment;
- Provides batch and stream analytics through dual versioning, smart data management, and memory optimization techniques;
- Supports diverse data access of various usecases with GraphView and data visibility abstractions.

The rest of the paper is organized as follows. We present a usecase in §2, opportunities and GRAPHONE overview in §3, the hybrid store in §4, data management internals and optimizations in §5, GraphView data abstraction in §6, evaluations in §7, related work in §8, and conclusion in §9.

## 2 Use Case: Network Analysis

Graph analytics is a natural choice for data analysis on an enterprise network. Fig. 2(a) shows a graph representation of a simple computer network. Such a network can be analyzed in its entirety by calculating the diameter [48], and betweenness centrality [13] to identify the articulation points. This kind of batch analysis is very useful for network infrastructure management. In the meantime, as the dynamic data flow within the network captures the real-time behaviors of the users and machines, the stream analytics is used to identify security risks, e.g., denial of service, and lateral movement, which can be expressed in the form of path queries, parallel paths and tree queries on a streaming graph [38, 18].

Los Alamos Nation Laboratory (LANL) recently released a comprehensive data set [37] that captures a wide range of network information, including authentication events, process events, DNS lookups, and network flows. The LANL data covers over 1.5 billion events, 12,000 users, and 17,000 computers, and spans 58 consecutive days. For example, the network authentication data captures the login information that a user logs in to a network machine, and also from that machine to other machines. When the network defense system identifies a malicious user and node, it needs to find all the nodes that may have been infected. Instead of analyzing every node of the network, one can quickly run a path traver-

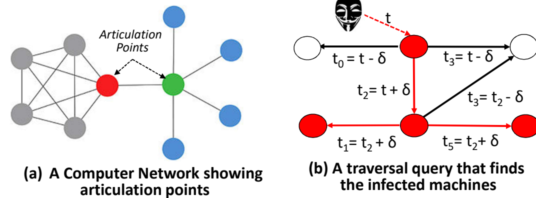


**Fig. 2:** Graph traversal can locate possible infected nodes using real-time authentication graph if infected user and node are known

sal query on the real-time authentication graph to identify the possible infected nodes, that is, find all the nodes whose login has originated from the chain of nodes that are logged in from the first infected machine [38] as shown in Fig. 2(b).

In summary, a high-performance graph store that captures dynamic data in the network, combined with user, machine information and network topology, is advantageous in understanding the health of the network, accelerating network service, and protecting it against various attacks. This work presents a graph storage and APIs for such usecases.

## 3 Opportunities and Overview

A graph can be defined as G = (V, E, W), where V is the vertex set, and E is the edge set, and W is the set of edge weights. Each vertex may also have a label. In this section, graph formats and their traits are described as relevant for GRAPHONE, and then we present its high-level overview.

### 3.1 Graph Representation: Opportunities

Fig. 3 shows three most popular data formats for a sample graph. First, the *edge list* is a collection of edges, a pair of vertices, and captures the incoming data in their arrival order. Second, the *compressed sparse row (CSR)* groups the edges of a vertex in an *edge array*. There is a metadata structure, *vertex array*, that contains the index of the first edge of each vertex. Third, the *adjacency list* manages the neighbors of each vertex in separate per-vertex edge arrays, and the vertex array stores a count (called degree) and pointer to indicate the length and the location of the corresponding edge arrays respectively. This format is better than the CSR for ingesting graph updates as it affects only one edge array at a time.

In the edge list, the neighbors of each vertex are scattered across, thus is not the optimal choice for many graph queries and batch analytics who prefer to get the neighboring edges of a vertex quickly [34, 29, 30, 12] etc . On the other hand, the adjacency list format loses the temporal ordering as the incoming updates get scattered over the edge arrays, thus not suited for stream analytics. Given their advantages and disadvantages, neither format is ideally suited for supporting both batch and stream analytics on its own. We now identify two opportunities for this work:

**Opportunity #1: Utilize both the edge list and the adjacency list within a hybrid store**. The edge list format preserves the data arrival order and offers a good support for fast updates as each update is simply appended to the end of the list. On the other hand, the adjacency list keeps all the neighbors of a vertex indexed by the source vertex, which provides efficient data access for graph analytics. Thus it allows GRAPHONE to achieve high-performance graph computation while simultaneously supporting fine-grained updates.

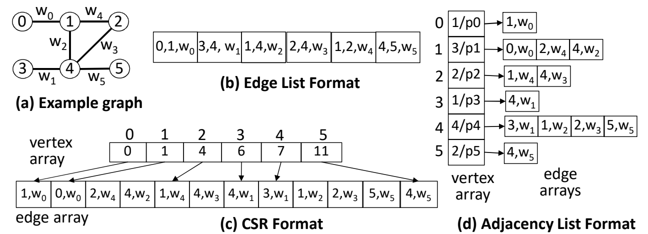


**Fig. 3:** Sample graph and its various storage format

**Opportunity #2: Fine-grained snapshot creation with the edge list format**. Graph analytics and queries require an immutable snapshot of the latest data for the duration of their execution. The edge list format provides a natural support for fine-grained snapshot creation without creating a physical snapshot due to its temporal nature, as tracking a snapshot is just remembering an offset in the edge list. Meanwhile, the adjacency list format through its coarse-grained snapshot capability [54, 26] is used to complement the edge list.

### 3.2 Overview

GRAPHONE utilizes a hybrid graph data store (discussed in §4) that consists of a small circular *edge log* and the *adjacency store*. Fig. 4 shows an high-level overview of GRAPHONE architecture. The hybrid store is managed in several phases (presented in §5). Specifically, during the *logging phase*, the edge log records the incoming updates in the edge list format in their arrival order, and supports a high ingestion rate. We define *non-archived edges* as the edges in the edge log that are yet to be moved to the adjacency store. When their number crosses the *archiving threshold*, a parallel *archiving phase* begins, which merges the latest edges to the adjacency store to create a new adjacency list snapshot. This duration is referred to as an *epoch*. In the *durable phase*, the edge log is written to a disk.

To efficiently create and manage immutable versions for data analytics in presence of the incoming updates, we provide a set of GraphView APIs (discussed in §6). Specifically, *static view* API is for batch processing, while *stream view* API is for stream processing. Internally, the views utilize *dual versioning* technique where the versioning capability of both formats are exploited. For example, a real-time static view can be composed by using the latest coarse-grained version of the adjacency store, and the latest fine-grained version of non-archived edges.

It is important to note that the GraphView also provides analytics with the flexibility to trade-off the granularity of data visibility for better performance, e.g., the analytics that prefer running only on the latest adjacency list store will avoid the cost associated with the access of the latest edges from the non-archived edges.

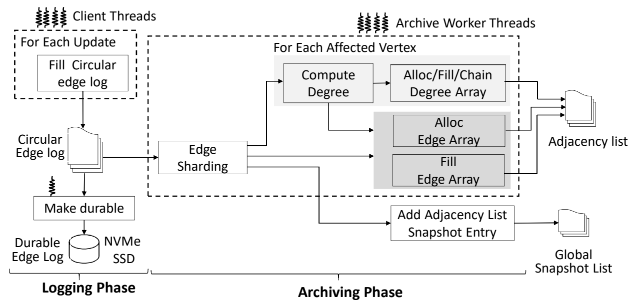


**Fig. 4:** Architecture of GRAPHONE. Operations related to same data structures have been grayed out in archiving phase. Compaction Phase is not shown.

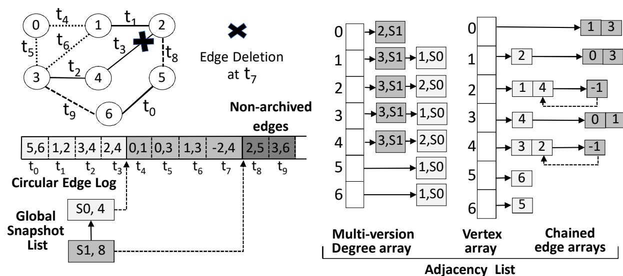


**Fig. 5:** The hybrid store for the data arrived from time $t_0$ to $t_9$: The vertex array contains pointers to the first and the last block of each edge array, while degree array contains deleted and added edge counts. However, only the pointer to the first block in the vertex array, and total count in the degree array are shown for brevity.

## 4 Hybrid Store

The hybrid store design presented in Fig. 5 consists of a small *circular edge log* that is used to record the latest updates in the edge list format. For deletion cases, we use tombstones, specifically the edge log also adds a new entry but the most significant bit (MSB) of the source vertex ID of the edge is set to denote its deletion as shown in Fig 5 for deleted edge $(2,4)$ at time $t_7$.

The *adjacency store* keeps the older data in the adjacency list format. The adjacency store is composed of *vertex array*, per-vertex *edge arrays*, and multi-versioned *degree array*. The *vertex array* contains a per-vertex flag and pointers to the first and last block of the edge arrays. Addition of a new vertex is done by setting a special bit in the per-vertex flag. Vertex deletion sets another bit in the same flag, and adds all of its edges as deleted edges to the edge log. These bits help GRAPHONE in garbage collecting the deleted vertex ID.

The *edge array* contains per-vertex edges of the adjacency list. It may contain many small edge blocks, each of which contains a count of the edges in the block and a memory pointer to the next block. The connection of edge blocks are referred to as *chaining*. An edge addition always happens at the end of the edge array of each vertex, which may require the allocation of a new edge block and linked to the last block. Fig. 5 shows chained edge arrays for the vertices with ID 1 to 4 for data updates that arrive in between $t_4$ to $t_7$. The adjacency list treats an edge deletion as an addition but the deleted edge entry in the edge array keeps the negative position of the original edge, while the actual data is not modified at all, as shown for edge $(2,4)$. As a result, deletion never breaks the convergence of a previous computation as it does not modify the dataset of the computation.

The *degree array* contains the count of neighboring edges of each vertex. Thus, a degree array from an older adjacency store snapshot can identify the edges to be accessed even from the latest edge arrays due to the latter's append-only property. Hence, the degree array in GRAPHONE is *multi-versioned* to support adjacency store snapshots. It keeps the total added and deleted edge counts of each vertex. Both counts help in efficiently getting the valid neighboring edges, as a client can do the exact memory allocation (refer to the *get-nebrs-*()* API in Table 2). When an edge is added or deleted for a vertex, a new entry is added for this vertex in the degree array in each epoch. Two different versions $S0$ and $S1$ of the degree array are shown in Fig. 5 for two epochs $t_0 - t_3$ and $t_4 - t_7$.

One can note that degree nodes are shared across epochs if there is no later activity in a vertex. For example, the same degree nodes for vertices with ID 5 and 6 are valid for both epochs in Fig. 5. The degree array nodes of an older versions may be garbage collected when the corresponding adjacency store snapshot retires, i.e., not being used actively by any analytics, and is tracked using reference counting mechanism through the global snapshot list, which will be discussed shortly. For example, if snapshot S0 is retired, then the degree nodes of snapshot S0 for vertices with ID $1-4$ can be reused by later snapshots (e.g. S2).

The *global snapshot list* is a linked list of snapshot objects to manage the relationship between the edge log and adjacency store at each epoch. Each node contains an absolute offset to the edge log where the adjacency list snapshot is created, and a reference count to capture the number of views using this adjacency list snapshot. A new entry in the global snapshot list is created after each epoch, and it implies that the edge log data of the last epoch has been moved to the adjacency store atomically, and is now visible to the world.

**Weighted Graphs**. Edge weights are generally embedded in the edge arrays along with the destination vertex ID. Some graphs have static weights, e.g., an edge weight in an enterprise network can represent the network speed between the two nodes. A weight change is then treated internally as an edge deletion followed by an edge addition. On the other hand, if edge weights are dynamic, such as network data flow, then such weights are suited for various analytics if kept for a configurable time window, e.g., anomaly detection in the network flow. In this case GRAPHONE is configured to treat weight changes as a new edge to aid such analytics.

**Dual Versioning and Data Overlap** GRAPHONE uses *dual versioning* to create the instantaneous read-only graph views (snapshot isolation) for data analytics. It exploits both the fine-grained versioning property of the edge log, and the

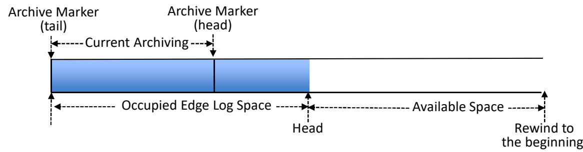


**Fig. 6:** Circular Edge log design showing various offset or markers. Markers for durable phase are similar to archiving and are omitted.

coarse-grained versioning capability of the adjacency list format. It should be noted that the adjacency list provides one version per epoch, while the edge log supports multiple versions per epoch, as many as the number of edges arrived during the epoch. So the dual versioning provides many versions within an epoch which is the basis for static views, and should not be confused with the adjacency list snapshots. In Fig. 5, static view at the time $t_6$ would be adjacency list snapshot $S0$ plus the edges from $t_4 - t_6$.

A small amount of *data overlap* between the two stores keeps the composition of the view intact. This makes the view accessible even when the edge log data is moved to the adjacency store to create a new adjacency list version. Thus both stores have the copy of a few epochs of the same data. For one or more long running iterative analytics, we may use the durable edge log or a private copy of non-archived edges to provide data overlap, so that analytics can avoid interfering with data management operations of the edge log.

# 5 Data Management and Optimizations

Data management faces the key issues of minimizing the size of non-archived edges, providing atomic updates, data ordering, and cleaning of older snapshots. Addition and deletion of vertices and edges, and edge weight modification are all considered as an atomic update.

## 5.1 Data Management Phases

Fig. 4 depicts the internals of the data management operations. It consists of four phases: *logging*, *archiving*, *durable* and *compaction*. Client threads send updates, and the logging to the edge log happens in the same thread context synchronously. The archiving phase moves the non-archived edges to the adjacency store using many worker threads, and one of them assumes the role of the master, called the archive thread. The durable phase happens in a separate thread, while compaction is multi-threaded but happens much later.

A client thread wakes up the archive thread and durable thread to start the archiving and durable phases when the number of non-archived edges crosses a threshold, called *archiving threshold*. The logging phase continues as usual in parallel to them. Also, the archive thread and durable thread check if any non-archived edges are there at the end of each phase to repeat their process, or wait for work with a timeout.

The edge log has a distinct offset or marker, *head*, for logging, which is incremented every time an edge is ingested as shown in Fig. 6. For archiving, GraphOne manages a pair of markers, i.e. the archiving operation happens from

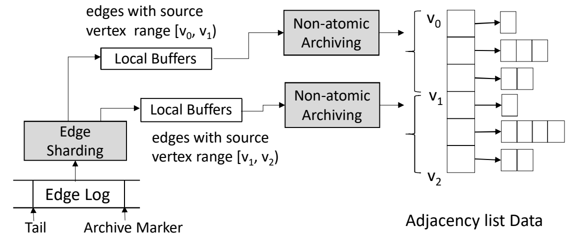


**Fig. 7:** Edge sharding separates the non-archived edges into many buffers based on their source vertex ID, so that the per-vertex edge arrays can keep the edge log arrival order, and enables non-atomic archiving.

the *tail archive marker* to the *head archive marker*, because the *head* will keep moving due to new updates. The durable phase also has a pair of markers to work with. Markers are always incremented and used with the modulo operator.

### 5.1.1 Logging Phase

The incoming update is converted to numerical identifiers, and acquires an edge list format. The mapping between vertex label to vertex ID and vice-versa manages this translation. Then a unique spot is claimed within the edge log by the atomic increment of the head, and the edge is written to a spot calculated using the modulo operation on the head, that also stores the operator (§4), addition or deletion, along with the edges. The atomicity of updates is ensured by the atomic increment of the head. The edge log is automatically reused in the logging phase due to its circular nature, and thus is overwritten by newer updates. Hence the logging may get blocked occasionally if the whole buffer is filled as the archiving or durable phases may not be able to catch up. We keep sufficiently large edge log to avoid frequent blocking. In case of blocked client threads, they are woken up when the archiving or durable phases complete.

### 5.1.2 Archiving Phase

This phase moves the non-archived edges from the edge log to the adjacency store. A naive multi-threaded archiving, where each worker can directly work on a portion of non-archived edges, may not keep the data ordering intact. If a deletion comes after the addition of an edge within the same epoch, the edge may become alive or dead in the edge arrays depending on the archiving order of the two data points.

An *edge sharding* stage in the archiving phase (Fig. 7) maintains per-vertex edges as per the edge log arrival to address the ordering problem. It shards the non-archived edges to multiple local buffers based on the range of their source vertex ID. For undirected graphs, the total edge count in the local buffer is twice of the non-archived edge count, as the ordering of reverse edges is also managed. For directed edges, both directions have their own local buffers.

The edges in each local buffer are then archived in parallel without using any atomic instructions. A heuristic is required for workload distribution, as the equal division is not possible among threads, thereby the last thread may get

more work assigned. To handle the workload imbalance among worker threads, we create a larger number of local buffers with smaller vertex range than the available threads, and assign different numbers of local buffers to each thread so that each gets an approximately equal number of edges to archive. The idea here is to assign slightly more than equal work to each thread, so that all the threads are balanced while the last thread is either balanced or lightly loaded.

This stage allocates new degree nodes or can reuse the same from the older degree array versions if they are not being used by any analytics. We follow these rules for reusing the degree array from older versions. We track the degree array usage by analytics using reference counting per epoch [40], and can be reused if all static views created within that epoch have expired, i.e., the references are dropped to zero (not being used by any running analytics). It also ensures that a newly created view uses the latest adjacency list snapshot that should never be freed.

The stage then populates the degree array, and allocates memory for edge blocks that are chained before filling those blocks. We then create a new snapshot object, fill it up with relevant details, and add it atomically to the global snapshot list. At the end of the archiving phase, the archive thread sets the tail archive marker atomically to the value of the head archive marker, and wakes up any the blocked client threads.

#### 5.1.3 Durable Phase and Recovery

The edge log data is periodically appended to a durable file in a separate thread context instead of logging immediately to the disk to avoid the overhead of IO system calls during each edge arrival. Also this will not guarantee durability unless fsync() is called. The logging uses buffered sequential write, and allows the buffer cache to work as spillover buffer for the access of non-archived edges if the edge log is over-written.

The durable edge log is a prefix of the whole ingested data, so GRAPHONE may lose some recent data in the case of an unplanned shutdown. The recovery depends on upstream backup that keep the latest data for some time, such as kafka [42], and replays it for the lost data, and creates the adjacency list on the whole data. Recovery is faster than building the data structures at an edge level, as only the archiving phase is involved working on bulk of data. Alternatively, persistent memory may be used for the edge log to provide durability at each update [45].

The durable phase also performs an incremental checkpointing of the adjacency store data from an old time-window, and frees the memory associated with it. This is useful for streaming data such as LANL network flow, where the old adjacency data can be checkpointed in disk, as the in-memory adjacency store within the latest time window is sufficient for stream analytics. By default, it is not enabled. During checkpointing the adjacency store, the vertex ID and length of the edge array are persisted along with edge arrays so that data can be read easily later, if required.

#### 5.1.4 Compaction Phase

The compaction of the edge arrays removes deleted data from per-vertex edge array blocks up to the latest retired snapshot identified via the reference counting scheme discussed in §5.1.2. The compaction needs a similar reference counting for the private static views (§6.1). For each vertex, it allocates new edge array block and copies valid data up to the latest retired snapshot from the edge arrays, and creates a link to the rest of the original edge array blocks. The newly created edge array block is then atomically replaced in the vertex array, while freeing happens later to ensure that cached references of the older data are dropped. This phase is generally clubbed with archiving phase where the degree array is updated to reflect the new combination.

### 5.2 Memory Overhead and Optimizations

The edge log and degree array are responsible for versioning. The edge log size is relatively small as it contains only the latest updates which moves quickly to the base store, e.g, the archiving threshold of $2^{16}$ edges translates to only 1MB for a plain graph assuming 8 byte vertex ID. Thus the edge log is only several MBs. The memory in degree arrays are also reused ( §5.1.2). This leaves us to memory analysis of edge arrays which may consume a lot of memory due to excessive chaining in their edge blocks. For example, GRAPHONE runs archiving phase for $2^{16}$ times for Kron-28 graph if the archiving threshold is $2^{16}$. In this case, the edge arrays would consume 148.73GB memory and have average 29.18 chain per-vertex. We will discuss the graph datasets used in this paper shortly. If all the edges were to be ingested in one archiving phase, this static system needs only an average 0.45 chain and 33.80GB memory. The chain count is less than one as 55% vertices do not have any neighbor.

GRAPHONE uses two memory allocation techniques, as we discuss next, to reduce the level of chaining to make the memory overhead of edge arrays modest compared to a static engine. The techniques work proactively, and do not affect the adjacency list versioning. Compaction further reduces the memory overhead to bring GRAPHONE at par with static analytics engine, but is performed less frequently.

**Optimization #1: Cacheline Sized Memory Allocation**. Multiples of cacheline sized memory is allocated for the edge blocks. One cacheline (64 bytes) can store up to 12 neighbors for the plain graph of 32bit type, leaving the rest of the space for storing a count to track space usage in the block and a link to the next block. In this allocation method, the ma-

Table 1: Impact of two optimizations on the chain count and memory consumption on the kronecker graph.

| Optimizations | Chain Count | | Memory Needed (GB) |
|---|---|---|---|
| | Average | Maximum | |
| Baseline System | 29.18 | 65,536 | 148.73 |
| +Cacheline memory | 2.96 | 65,536 | 47.42 |
| +Hub Vertex Handling | 2.47 | 3,998 | 45.79 |
| Static System | 0.45 | 1 | 33.81 |

Table 2: Basic GraphView APIs

| Return | API |
|---|---|
| **Static View APIs** | |
| snap-handle | create-static-view(global-data, simple, private, stale) |
| status | delete-static-view(snap-handle) |
| count | get-nebr-length-{in/out}(snap-handle, vertex-id) |
| count | get-nebrs-{in/out}(snap-handle, vertex-id, ptr) |
| count | get-nebrs-archived-{in/out}(snap-handle, vertex-id, ptr) |
| count | get-non-archived-edges(snap-handle, ptr) |
| **Stateless Stream View APIs** | |
| stream-handle | reg-stream-view(global-data, window-sz, batch-sz) |
| status | update-stream-view(stream-handle) |
| status | unreg-stream-view(stream-handle) |
| count | get-new-edges-length(stream-handle) |
| count | get-new-edges(stream-handle, ptr) |
| **Stateful Stream View APIs** | |
| sstream-handle | reg-sstream-view(global-data, window-sz, v-or-e-centric, simple, private, stale) |
| status | update-sstream-view(sstream-handle) |
| status | unreg-sstream-view(sstream-handle) |
| bool | has-vertex-changed(sstream-handle, vertex-id) |
| count | get-nebr-length-{in/out}(sstream-handle, vertex-id) |
| count | get-nebrs-{in/out}(sstream-handle, vertex-id, ptr) |
| count | get-nebrs-archived-{in/out}(sstream-handle, vertex-id, ptr) |
| count | get-non-archived-edges(sstream-handle, ptr) |
| **Historic View APIs** | |
| count | get-prior-edges(global-data, start, end, ptr) |

jority of the vertices will need only a few levels of chaining. For example, in a Twitter graph, 88.43% of the vertices will need at most 3 cachelines only, and so do 92.49% for Kron-28 graph. This optimization reduces the average chain count by 9.88×, and memory consumption by 3.14× in comparison to a baseline system as shown in Table 1. The baseline system uses a dynamic block size which is equivalent to the number of edges arrived during each epoch for each vertex.

**Optimization #2: Hub Vertex Handling**. A few vertices, called hub-vertices, have very high degree in a graph that follows power-law distribution [22]. They are very common in real-life graphs, such as for the twitter follower graph whose degree distribution we analyze. Such vertices are likely to participate in each archiving phase. Hence they will have a lot of chaining in their edge arrays, and the aforementioned memory management technique alone is not enough. In this case, we allocate in multiples of 4KB page-aligned memory for vertices that already have 8,192 edges or if the number of neighbors in any archiving phase crosses 256. The average chain count is further reduced to 2.47, leading to reduction in memory utilization by 1.63GB as listed in Table 1. One can vary the threshold to identify a hub vertex but performance remains similar to the cacheline sized memory (Fig. 15).

## 6 GraphView Abstraction

GraphView data abstraction hides the complexity of the hybrid store by providing simple data access APIs as shown in Table 2. The *static view* is suited for batch analytics and queries, while the *stream view* for stream processing. Both offer diverse data access at two granularities of data visibility of updates. At any time, a number of views may co-exist without incurring much memory overhead, as the view data

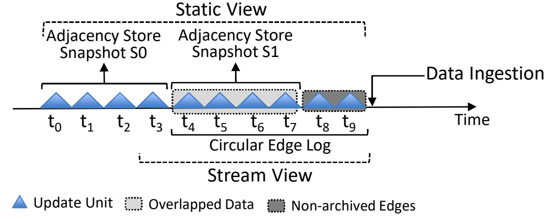


**Fig. 8:** GRAPHONE hybrid store illustrating various views with two adjacency store versions, S0 and S1, with a small edge log

is composed of the same adjacency store and non-archived edges as shown in Fig. 8. The access of non-archived edges provides data visibility at the edge level granularity.

Due to the cost of indexing the non-archived edges, GraphView provides an option to trade-off the granularity of data visibility to gain performance. Further, one can use vertex-centric compute model [73] on the adjacency list plus edge-centric compute model [81, 43, 66] on non-archived edges, so there is no need to index the latter as plotted later to find its optimal minimum size (Fig. 13).

### 6.1 Static View

Batch analytics and queries prefer snapshots for computation, which can be created in real-time using *create-static-view()* API. It is represented by an opaque handle that identifies the view composition, i.e., the non-archived edges and the latest adjacency list snapshot, and serves as input to other static view APIs. A created handle should be destroyed using *delete-static-view()*. Based on the input supplied to *create-static-view()* API, many types of static view are defined.

**Basic Static View**. This view is very useful for advanced users and higher level library development which prefer more control and performance. The main low-level API are: *get-nebrs-archived-*()* that returns the reference to the per-vertex edge array; and *get-non-archived-edges()* that returns the non-archived edges. On the other hand, it also provides a high-level API, *get-nebrs-*()*, that returns the neighbor list of a vertex by combining the adjacency store and the non-archived edges in a user supplied memory buffer. It may be preferable by queries with high selectivity that only need to scan the non-archived edges for one or a few vertex, e.g. 1-hop query, and is not apt for long running analytics.

The implementation of *get-nebrs()* for the non-deletion case is a simple two step process: copy the per-vertex edge array to the user supplied buffer, followed by a scan of the non-archived edges to find and add the rest of the edges of the vertex to the buffer. For the deletion case, both the steps track the deleted positions in the edge arrays, and the last few edges from edge arrays and/or non-archived edge log are copied into those indexes of the buffer.

**Private Static View**. For long running analytics, keeping basic static views accessible have some undesirable impacts: (1) all the static views may have to use the durable edge log if the corresponding non-archived edges in the edge log has been overwritten; (2) the degree array cannot be reused in

**Algorithm 1** Traditional BFS using static view APIs

```
1:  handle ← create-static-view(global-data, private=true, simple=true)
2:  level = 1; active-vertex = 1; status-array[root-vertex] = level;
3:  while active-vertex do
4:      active-vertex = 0;
5:      for vertex-type v = 0; v < vertex-count; v++ do
6:          if status-array[v] == level then
7:              degree ← get-nebrs-out(handle, v, nebr-list);
8:              for j=0; j < degree; j++ do
9:                  w ← nebr-list[j];
10:                 if status-array[w] == 0 then
11:                     status-array[w] ← level + 1; ++active-vertex;
12:     ++level;
13: delete-static-view(handle)
```

the archiving phase as it is still in use. To solve this, one can create a *private static view* by passing *private=true* in the *create-static-view()* API. In this case, a private copy of the non-archived edges and the degree array are kept inside the view handle with their global references dropped to make it independent from archiving. One can pass *simple=true* in the *create-static-view()* to create a temporary in-memory adjacency list from the non-archived edges for optimizing get-nebr-*() API, as shown in Algorithm 1 for a simplified BFS (push model) implementation. This approach is more flexible than static analytics engine which converts the whole data, or dynamic graph system that disallows the user to choose fine-grained control on snapshot creation.

Creation to many private static views may introduce memory overhead. To avoid this, a reference of the private degree array is kept in the snapshot object and is shared by other static views created within that epoch, and are locally reference counted for freeing. Thus, creating many private views within an epoch has overhead of just one degree array. However, creating many private static views across epochs may still cause the memory overhead, if older views are still being accessed by long running analytics. This also means that the machine is overloaded with computations, and they are not real-time in nature. In such a case, a user may prefer to copy the data to another machine to execute them.

**Stale Static View**. Many analytics are fine with data visibility at coarse-grained ingestions, thus some stale but consistent view of the data may be better for their performance. In this case, passing *stale=true* returns the snapshot of the latest adjacency list only. This view can be combined with private static view where degree array will be copied.

## 6.2 Stream View

Stream computations follow a pull method in GraphOne, i.e., the analytics pulls new data at the end of incremental compute to perform the next phase of incremental compute. The stream view APIs around the handle simplify the data access and its granularity in presence of the data ingestion. Also, checkpointing the computation results and the associated data offset is the responsibility of the stream engine, so that the long running computation can be resumed from that point onwards in case of a fault.

**Algorithm 2** A stateless stream compute skeleton

```
1: handle ← reg-stream-view(global-data, batch-sz=10s)
2: init-stream-compute(handle)                      ▷ Application specific
3: while true do                                    ▷ Or application specific criteria
4:     if update-stream-view(handle) then
5:         count = get-new-edges(handle, new-edges)
6:         for j=0; j < count; j++ do
7:             do-stream-compute(handle, new-edges[j]) ▷ Or any method
8: unreg-sstream-view(handle)
```

**Stateless Stream Processing**. A stateless computation, e.g. counting incoming edges (aggregation), only needs a batch of new edges. It can be registered using the *reg-stream-view()* API, and the returned handle contains the batch of new edges. Algorithm 2 shows how one can use the API to do stateless stream computation. The handle also allows a pointer to point to analytics results to be maintained by the stream compute implementation. The implementation also needs to checkpoint only the edge log offset and the computation results as GraphOne keeps the edge log durable.

An extension of the model is to process on a data window instead on the whole arrived data. For sliding window implementation, GraphOne manages a cached batch of edge data around the start marker of the data window in addition to the batch of new edges. The old cached data can be accessed by the analytics for updating the compute results, e.g., subtracting the value in aggregation over the data window. The cached data is fetched from the durable edge log, and shows sequential read due to the sliding nature of the window. A tumbling window implementation is also possible where the batch size of new edges is equal to the window size, and hence does not require older data to be cached. Additional checkpointing of the starting edge offset is required along with the edge log offset and computation results.

**Stateful Stream Processing**. A complex computation, such as graph coloring [67], is stateful that needs the streaming data and complete base store to access the computational state of the neighbors of each vertex. A variant of static view is better suited for it because its per-vertex neighbor information eases the access of the computational state of neighbors. It is registered using *reg-sstream-view()*, and returns *sstream-handle*. For edge-centric computation, the handle also contains a batch of edges to identify the changed edges. For vertex-centric computation, the handle contains per-vertex one-bit status to denote the vertex with edge updates that can be identifies using the *has-vertex-changed()* API. This is updated during *update-sstream-view()* call that also updates the degree array. Algorithm 3 shows an example code snippet.

As the degree array plays an important role for a stateful computation due to its association with the static view, using an additional degree array at the start marker of the data window eases the access of the data within the window from the adjacency store. The *sstream-handle* manages the degree array on behalf of the stream engine, and internally keeps a batch of cached edges around the start marker of the window

**Algorithm 3** A stateful stream compute (vertex-centric) skeleton

```
1: handle ← reg-sstream-view(global-data, v-centric, stale=true)
2: init-sstream-compute(handle)                ▷ Application specific
3: while true do                      ▷ Or application specific criteria
4:   if update-sstream-view(handle) then
5:     for v=0; v < vertex-count; v++ do
6:       if has-vertex-changed(handle, v) then
7:         do-sstream-compute(handle, v)    ▷ Application specific
8: unreg-sstream-view(handle)
```

to update the old degree array. The *get-nebrs-*()* function returns the required neighbors only. Checkpointing the computational results, the edge log offset at the point of computation, and window information is sufficient for recovery.

### 6.3 Historic Views

GRAPHONE provides many views from recent past, but it is not designed for getting arbitrary historic views from the adjacency store. However, durable edge log can provide the same using *get-prior-edges()* API in edge list format as it keeps deleted data, behaving similar to existing data stores [14, 23]. Moreover, in case of no deletion, one can create a degree array at a durable edge log offset by scanning the durable edge log, and the degree array will serve older static or stream view from the adjacency store to gain insights from the historical data. For data access from a historical time-window in this case, one need to build two degrees arrays at both the offsets of the durable edge log.

## 7 Evaluations

GRAPHONE is implemented in around 16,000 lines of C++ code including various analytics. It supports plain graphs and weighted graphs with either 4 byte or 8 byte vertex sizes. We store the fixed weights along with the edges, variable length weights in a separate weight store using indirection. Any type of value can be stored in place of weight such as integers, float/double, timestamps, edge-id or any custom weight as the code is written using C++ templates. So one can write a small plug-in describing the weight structures and other functions, and GRAPHONE would be ready to serve a custom weight. All experiments are run on a machine with 2 Intel Xeon CPU E5-2683 sockets, each having 14 cores with hyper-threading enabled. It has 512GB memory, Samsung NVMe 950 Pro 512GB, and CentOS 7.2. Prior results have also been performed on the same machine.

We choose data ingestion, BFS, PageRank and 1-Hop query to simulate the various real-time usecases to demonstrate the impact of GRAPHONE on analytics. BFS and PageRank are selected because many real-time analytics are iterative in nature, e.g. shortest path, and many prior graph systems readily implement them for comparison. 1-Hop query accesses the edges of random 512 non-zero degree vertices and sums them up to make sure we access them all. 1-Hop query simulates many small query usecases, such as listing one's friends, or triangle completion to get friend suggestions in a social graph, etc. During the ingestion, vertex name to vertex ID conversion was not needed as we directly used the vertex ID supplied with these datasets as followed by other graph systems. All the edges will be stored twice in the adjacency list: in-edges and out-edges for directed graphs, and symmetric edges for undirected graphs. No compaction was running in any experiments unless mentioned.

**Datasets**. Table 3 lists the graph datasets. Twitter [3], Friendster [1] and Subdomain [4] are real-world graphs, while Kron-28 and Kron-21 are synthetic kronecker graphs generated using graph500 generator [25], all with 4 byte vertex size and without any weights. LANL network flow dataset [74] is a weighted graph where vertex and weight sizes are 4 bytes and 32 bytes respectively, and weight changes are treated as new streaming data. We run experiment on first 10 days of data. We test deletions on a weighted RMAT graph [15] generated with [56] where vertex and weight sizes are 8 bytes. It contains 4 million vertices, and 64 million edges, and a update file containing 40 million edges out of which 2,501,937 edges are for deletions.

### 7.1 Data Ingestion Performance

**Logging and Archiving Rate**. Logging to edge log is naturally faster, while archiving rate depends upon the archiving threshold. Table 3 lists the logging rate of a thread, and archiving rate at the archiving threshold of $2^{16}$ edges for our graph dataset. A thread can log close to 80 million edges per second, while archiving rate is only around 45 million edges second at the archiving threshold for most of the graphs. Both the rates are lower for LANL graph, as the weight size is 32 bytes, while others have no weights.

**Ingestion Rate**. It is defined as single threaded ingestion to the edge log at one edge at a time, and leaving the archive thread and durable phase to automatically change with the arrival rate. The number is reported when all the data are in the adjacency store, and persisted in the NVMe ext4 file. GRAPHONE achieves an ingestion rate of more than 45 million edges per second, except LANL graph. The ingestion rate is higher than archiving rate (at the archiving threshold) except in Kron-21, as edges more than the archiving threshold are archived in each epoch due to higher logging rate. This indicates that GRAPHONE can support a higher arrival rate as archiving rate can dynamically boost with increased arrival velocity. The Kron-21 graph is very small graph, and the thread communication cost affects the ingestion rate.

**Compaction Rate**. We run compaction as a separate benchmark after all the data has been ingested. The graph compaction rate is 345.53 million edges per second for the RMAT graph which has more than 2.5 million deleted edges out of total 104 million edges. Results for other graphs are shown in Table 3. The poor rate for LANL graph is due to long tail for compacting edge arrays of few vertices. As shown later in Fig. 12, the compaction improves the analytics performance where the static GRAPHONE serves compacted adjacency list as it had no link in its edge arrays.

Table 3: Graph datasets showing vertex and edge counts in **millions** (M), and different rates in **millions edges/s** (M/s). The results show that the ingestion rate would be upper and lower bounded by the logging and archiving rate. D = Directed, U = Undirected. For deletions see §7.2.

| Graph Name | Type | Vertex Count (M) | Edge Count (M) | Individual Phases (M/s) | | In-Memory Rate (M/s) | | Ext-Memory Rate (M/s) | | Compaction Rate (M) |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | Logging | Archiving | Ingestion | Recovery | Ingestion | Recovery | |
| LANL | D | 0.16 | 1,521.19 | 35.98 | 28.91 | 26.99 | 30.23 | 25.26 | 29.48 | 41.85 |
| Twitter | D | 52.58 | 1,963.26 | 82.62 | 47.98 | 66.39 | 71.28 | 61.13 | 71.87 | 541.71 |
| Friendster | D | 68.35 | 2,586.15 | 82.85 | 49.32 | 60.40 | 95.78 | 58.35 | 95.44 | 520.65 |
| Subdomain | D | 101.72 | 2,043.20 | 82.86 | 43.43 | 68.25 | 180.75 | 61.54 | 151.96 | 444.84 |
| Kron-28 | U | 256 | 4,096 | 79.23 | 43.68 | 52.39 | 116.18 | 49.70 | 107.61 | 798.91 |
| Kron-21 | U | 2 | 32 | 78.91 | 78.40 | 58.31 | 90.44 | 57.02 | 66.66 | 1011.68 |

**Durability**. The durable phase has less than 10% impact on the ingestion rate. Table 3 shows the in-memory ingestion rate and can be compared against that of GraphOne, which uses NVMe SSD for durability. This is because durable phase runs in a separate thread context, and exhibits only sequential write. The NVMe SSD can support up to 1500MB/s sequential write and that is sufficient for GraphOne as it only needs smaller write IO throughput, as shown in Fig. 9 for Friendster graph. This indicates that a higher logging rate can easily be supported by using a NVMe SSD.

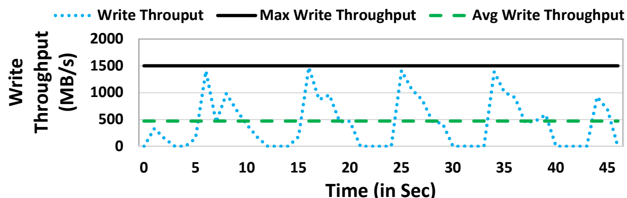


**Fig. 9:** Write throughput for friendster in GraphOne comparing against average requirement and maximum available in an NVMe

**Recovery**. Recovery only needs to perform archiving phase at bulk of data. As we will show later in Fig. 13, the archiving is fastest when around $2^{27}$–$2^{31}$ edges are cleaned together. Hence we take the minimum of this size as recovery threshold to minimize the memory requirement of IO buffer and the recovery time, and also gets an opportunity to pipeline the IO read time of the data with recovery. Table 3 shows the total recovery time, including data read from NVMe SSD after dropping the buffer cache. Clearly, GraphOne hides the IO time when compared against in-memory recovery. The recovery rate varies a lot for different graph due to different distribution of the batch of graph data that has profound impact on parallelism and hence locality access of edge arrays.

## 7.2 Graph Systems Performance

We choose different classes of graph systems to compare against GraphOne. Stinger is a dynamic graph system, Neo4j and SQLite are graph databases, and Galois and static version of GraphOne are static graph systems. Except stream computations, all the analytics in this section are performed on private static view containing no non-archived edges as it is created at the end of the ingestion.

**Dynamic Graph System**. Stinger is an in-memory graph system that uses atomic instructions to support fine-grained updates. So it cannot provide semantically correct analytics if updates and computations are scheduled at the same time, as different iteration of the analytics will run on the different versions of the data. We used the benchmark developed in [56] to compare the results on the RMAT graph.

Stinger is able to support 3.49 million updates/sec on the same weighted RMAT graph, whereas GraphOne ingests 18.67 million edges/sec, achieving 5.36× higher ingestion rate. Part of the reason for poor update rate of Stinger is that unlike GraphOne, it directly updates the adjacency store using atomic constructs. We have implemented PageRank and BFS in a similar approach as Stinger. The comparison is plotted in Fig. 10. Clearly, GraphOne is able to provide a better support for BFS and PageRank achieving 12.76× and 3.18× speedup respectively. The reason behind the speedup is explicit optimization to reduce the chaining which removes a lot of pointer chasing, and better cache access locality due to cacheline sized edge blocks.

**Fig. 10:** Comparison against Stinger for in-memory setup

**Databases**. We compare against SQLite 3.7.15.2, a relational database, Neo4j 3.2.3, a graph database for ingestion test. SQLite and Neo4j support ACID transaction, and do not provide native support for graph analytics. It is known that higher update rate is possible by trading off the strict serializability of databases, however to measure the magnitude of improvement, it is necessary to conduct experiment.

The in-memory configuration of SQLite can ingest 12.46K edges per second, while GraphOne is able to support 18.67 million edges per second in the same configuration for above dataset. Neo4j could not finish the benchmark after more than 12 hours, which is along the same line as observed in [56]. Hence we have tested on a smaller graph with 32K vertices, 256K edges, and 100K updates. Neo4j is configured to use disk to make it durable. Neo4j and GraphOne both use the buffer cache while persisting the graph data. Neo4j can ingest only 14.81K edges per second, whereas GraphOne ingests at 3.63M edges per second.

**Static Graph System**. We compare against Galois, a representative in-memory static graph engine based on CSR format. It does not provide the data management capabil-

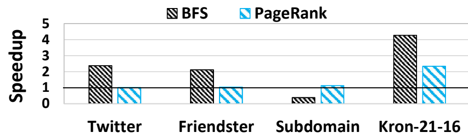


**Fig. 11:** Speedup comparison of GRAPHONE with Galois (pre-processing cost not included).

ity, so the whole graph is constructed in one time, called pre-processing time, which takes a significant amount of time [55]. In contrast, GRAPHONE can start the analytics without any pre-processing. Fig. 11 shows the speed up of GRAPHONE for PageRank and BFS over Galois (without pre-processing cost) for all the graphs except Kron-28 as Galois had a memory error. The PageRank results are almost same as it is compute intensive, thus effect of chaining is not observed. For Kron-21-16 which is very small, the performance of Galois is bad. We suspect that the cost of manual workload division in Galois for small graphs affects its performance, while we use dynamic scheduling of OpenMP.

For BFS, GRAPHONE performs better than Galois with an exception in the Subdomain graph. Both systems have same BFS implementation (direction-optimized BFS [11]) with a minor implementation difference. Our BFS is implemented using the status array metadata where the level of each vertex is maintained as one byte word, and tracking the active vertices requires revisiting whole status array. Galois uses the frontier queue where active vertices are kept in a separate work queue. Based on our experience with graph systems, status array implementation is faster for small diameter graphs, otherwise frontier queue approach is better. The Subdomain graph has 140 BFS levels (the highest of all graphs) hence we perform poorly, but Kron-21 has only 7 levels (the least of all the graphs) so the speedup is the highest.

**Static GRAPHONE**. GRAPHONE is expected to perform slightly worse than the static graph engine without including the pre-processing cost, but much better if including. Therefore to demonstrate the performance overhead of data management and chaining without any specific algorithm differences, we compare GRAPHONE against the static configuration of itself where maximum chain count is just one.

Fig. 12 shows this performance drop (without including pre-processing cost), specifically trading off just 17% average performance for real-world graphs (26% for all the graphs plotted) from the static system, one can support high arrival velocity of fine-grained updates. However, the performance drop is only temporary as the compaction process will remove the chaining in the background. Moreover, when adding the pre-processing cost to the static system, GRAPHONE is able to perform better. For example, the pre-processing cost for Kron-28 graph is 32.73s, one or multiple orders of magnitude longer than the runtime of these algorithms, e.g. 34.12× more than the run-time of BFS.

**Stream Graph Engines**. The logging and archiving operations are examples of different categories of stream analytics: logging is a proxy to continuous stateless stream analytics, while archiving is same to the discrete stateful stream analytics. Thus, Table 3 is also an indication of their performance. We have also implemented a streaming weakly connected components using ideas from COST [57] using stateless stream view APIs and it can process 33.60 million stream edges/s on Kron-28 graph.

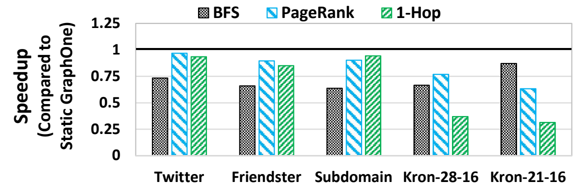


**Fig. 12:** Graph analytics performance in GRAPHONE compared to its static version that have no chaining requirement.

GRAPHONE runs stream computation and data ingestion concurrently, while prior stream processing systems interleave them that results into lower ingestion rate. To demonstrate the advantage of this design decision, we have implemented a streaming PageRank using stateful stream view APIs that runs in parallel to data ingestion in GRAPHONE. To simulate the prior stream processing we interleave the two. The execution shows that GRAPHONE improves the data ingestion by 26.22% for Kron-28 graph. We leave the comparison against other stream processing engine as future work as the focus of this work is on graph data-store.

## 7.3 System Design Parameters

**Performance Trade-off in Hybrid Store**. We first characterize the behavior of the hybrid store for different number of non-archived edges. Fig. 13 shows the performance variation of archiving rate, BFS, PageRank, and 1-hop query for Kron-28 graph when the non-archived edge counts are increased, while the rest of the edges are kept in the adjacency store for Kron-28. The figure shows that up to $2^{17}$ non-archived edges in the edge log brings negligible drop in the analytics performance. Hence, we recommend the value of archiving threshold as $2^{16}$ edges as the logging overlaps with the archiving. GRAPHONE is able to sustain an archiving rate 43.68 million edges per second at this threshold. The 1-Hop query latency of all 512 queries together is only 53.766 ms, i.e. 0.105 ms for each query.

The archiving threshold of $2^{16}$ edges is not unexpected as it is small compared to total edge count ($2^{33}$) in Kron-28, and

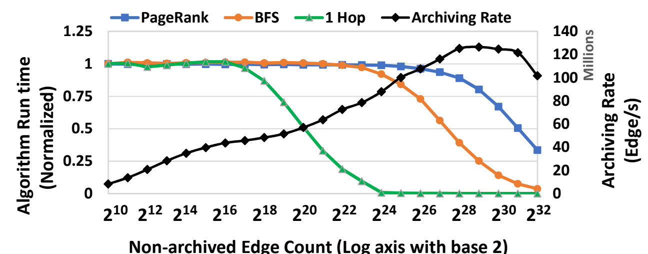


**Fig. 13:** Algorithmic performance and archiving rate variation for different non-archived edge count

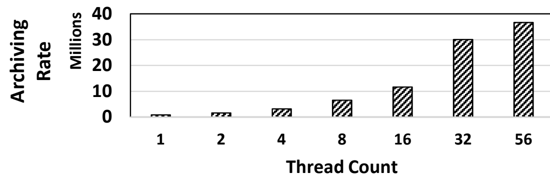

**Fig. 14:** Archiving rate scaling with thread count

the analytics on non-archived edges are parallelized. Further, the parallelization cost dominates when the number of non-archived edges are small ($2^{10}$). Thus the analytics cost drops only when the number of non-archived edges becomes large.

Fig. 13 also shows that higher archiving threshold leads to better archiving rate, e.g., a archiving threshold of 1,048,576 ($2^{20}$) edges can sustain a archiving rate of 56.99 million edges/second. The drawback is that the analytics performance will be reduced as it will find more number of non-archived edges. On the contrary, archiving works continuously and tries to minimize the number of non-archived edges, so a smaller arrival rate will lead to frequent archiving, and thus fewer non-archived edges will be observed at any time. The drop in archiving rate at the tail is due to the impact of large working set size that leads to more last-level-cache transactions and misses while filling the edge arrays.

**Scalability**. The edge sharding stage removes the need of atomic instruction or locks completely in the archiving phase, which results into better scaling of archiving rate with increasing number of threads as plotted in Fig. 14. There is some super-linear behavior when thread count is increased from 16 to 32. This is due to the second socket coming into picture with its own hardware caches, and non-atomic behavior makes it to scale super-linear. This observation is confirmed by running the archiving using 16 threads spread equally across two sockets, and achieves higher archiving rate compared to the case when the majority of threads belong to one socket.

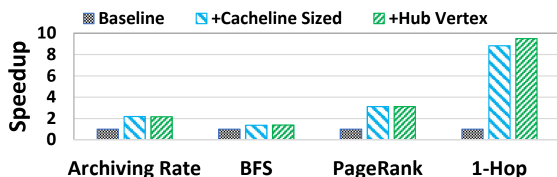

**Fig. 15:** Caheline sized memory allocation brings huge performance gain, while hub vertex handling on top of cacheline size memory allocation improves the query performance only.

**Memory Allocation**. Fig. 15 shows that the cacheline sized memory allocation and special handling of hub-vertices improve the performance of the archiving and analytics. The cacheline sized memory optimization improves the archiving rate at the archiving threshold by 2.20× for Kron-28 graph, while speeding up BFS, PageRank and 1-Hop query performance by 1.37×, 3.11× and 8.82×. Hub vertices handling additionally improves the query performance (by 7.5%).

**Edge Log Size**. Fig. 16 shows the effect of edge log size on overall ingestion rate on Kron-28 graph. Clearly, an edge log

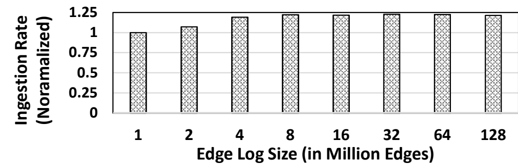

**Fig. 16:** Showing Ingestion rate when edge log size increases.

size greater than 4 million edges (32 MB) does not have any impact on overall ingestion rate.

## 8 Related Work

Static graph analytics systems [66, 50, 43, 60, 52, 79, 16, 36, 53, 65, 27, 44, 81, 28, 76, 7, 80] support only batch analytics where pre-processing consumes much more time than the computation itself [55]. Grapchi [47] and other snapshot based systems [35, 54, 62, 39, 26] support bulk updates only. Naiad [58], a timely dataflow framework, supports iterative and incremental compute but does not offer the data window on the graph data. Other stream analytics systems [17, 32, 58, 72] support stream processing and snapshot creation, some offering data window but all at bulk updates only. Stream databases [5, 6] provide only stream processing. TIDE [77] introduces probabilistic edge decay that samples data from base store.

Prior works [24, 69] follow integrated graph system model that manage online updates and queries in the database, and replicate data in an offline analytics engines for long running graph analytics tasks. As we have identified in §1, they suffers from excessive data duplication and weakest component problem. Zhang et al [78] also argue that such composite design is not optimal. GraPU [71] proposes to pre-processes the buffered updates instead of making them available to compute as in GraphOne. Trading-off granularity of data visibility is similar to Lazybase [19], but we additionally tune the access of non-archived edges to reduce performance drop in our setup and offer diverse data views.

The in-memory adjacency list in Neo4j [59] is optimized for read-only workloads, and new updates generally require invalidating and rebuilding those structures [63]. Titan [2], an open source graph analytics framework, is built on top of other storage engines such as HBase and BerkeleyDB, and thus does not offer adjacency list at the storage layer.

## 9 Conclusion

We have presented GraphOne, a unified graph data store abstraction that offers diverse data access at different granularity for various real-time analytics and queries at high-performance, while simultaneously supporting high arrival velocity of fine-grained updates.

## Acknowledgments

The authors thank the USENIX FAST'19 reviewers and our shepherd Ashvin Goel for their suggestions. This work was supported in part by National Science Foundation CAREER award 1350766 and grants 1618706 and 1717774.

## References


[1] Friendster Network Dataset – KONECT. http://konect.uni-koblenz.de/networks/friendster.

[2] Titan Graph Database. https://github.com/thinkaurelius/titan.

[3] Twitter (MPI) Network Dataset – KONECT. http://konect.uni-koblenz.de/networks/twitter_mpi.

[4] Web Graphs. http://webdatacommons.org/hyperlinkgraph/2012-08/download.html.

[5] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, et al. The Design of the Borealis Stream Processing Engine. In *Cidr*, volume 5, pages 277–289, 2005.

[6] D. J. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A New Model and Architecture for Data Stream Management. *VLDB Journal*, 12(2):12039, 2007.

[7] Z. Ai, M. Zhang, Y. Wu, X. Qian, K. Chen, and W. Zheng. Squeezing out All the Value of Loaded Data: An out-of-core Graph Processing System with Reduced Disk I/O. In *2017 USENIX Annual Technical Conference (USENIX ATC 17),(Santa Clara, CA)*, pages 125–137, 2017.

[8] L. Akoglu, H. Tong, and D. Koutra. Graph based anomaly detection and description: a survey. *Data Mining and Knowledge Discovery*, 29(3):626–688, May 2015.

[9] R. Albert, H. Jeong, and A.-L. Barabási. Internet: Diameter of the World-Wide Web. *Nature*, 401(6749):130–131, Sept. 1999.

[10] D. Anicic, P. Fodor, S. Rudolph, and N. Stojanovic. EP-SPARQL: A Unified Language for Event Processing and Stream Reasoning. In *Proceedings of the 20th international conference on World wide web*, pages 635–644. ACM, 2011.

[11] S. Beamer, K. Asanovic, and D. Patterson. Direction-Optimizing Breadth-First Search. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[12] B. Bhattarai, H. Liu, and H. H. Huang. CECI: Compact Embedding Cluster Index for Scalable Subgraph Matching. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, 2019.

[13] U. Brandes. A faster algorithm for betweenness centrality*. *Journal of Mathematical Sociology*, 25(2):163–177, 2001.

[14] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. C. Li, et al. TAO: Facebook's Distributed Data Store for the Social Graph. In *USENIX Annual Technical Conference*, pages 49–60, 2013.

[15] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A Recursive Model for Graph Mining. In *SDM*, 2004.

[16] R. Chen, J. Shi, Y. Chen, and H. Chen. PowerLyra: Differentiated Graph Computation and Partitioning on Skewed Graphs. In *Proceedings of the Tenth European Conference on Computer Systems*, 2015.

[17] 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 *Proceedings of the 7th ACM european conference on Computer Systems*, 2012.

[18] S. Choudhury, L. B. Holder, G. Chin, K. Agarwal, and J. Feo. A Selectivity based approach to Continuous Pattern Detection in Streaming Graphs. In *18th International Conference on Extending Database Technology (EDBT)*, 2015.

[19] J. Cipar, G. Ganger, K. Keeton, C. B. Morrey III, C. A. Soules, and A. Veitch. LazyBase: trading freshness for performance in a scalable database. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 169–182. ACM, 2012.

[20] D. Easley and J. Kleinberg. *Networks, crowds, and markets: Reasoning about a highly connected world*. Cambridge University Press, 2010.

[21] D. Ediger, R. McColl, J. Riedy, and D. A. Bader. Stinger: High Performance Data Structure for Streaming Graphs. In *High Performance Extreme Computing (HPEC), 2012 IEEE Conference on*, pages 1–5. IEEE, 2012.

[22] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On Power-Law Relationships of the Internet Topology. In *ACM SIGCOMM computer communication review*, volume 29, pages 251–262. ACM, 1999.

[23] FlockDB. https://blog.twitter.com/engineering/en_us/a/2010/introducing-flockdb.html, 2010.

[24] Graph Compute with Neo4j. https://neo4j.com/blog/graph-compute-neo4j-algorithms-spark-extensions/, 2016.

[25] Graph500. http://www.graph500.org/.

[26] 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 *EuroSys*, 2014.

[27] W.-S. Han, S. Lee, K. Park, J.-H. Lee, M.-S. Kim, J. Kim, and H. Yu. TurboGraph: A Fast Parallel Graph Engine Handling Billion-scale Graphs in a Single PC. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2013.

[28] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. Green-Marl: a DSL for Easy and Efficient Graph Analysis. In *ACM SIGARCH Computer Architecture News*, 2012.

[29] Y. Hu, P. Kumar, G. Swope, and H. H. Huang. Trix: Triangle Counting at Extreme Scale. In *High Performance Extreme Computing Conference (HPEC), 2017 IEEE*, pages 1–7. IEEE, 2017.

[30] Y. Hu, H. Liu, and H. H. Huang. TriCore: Parallel Triangle Counting on GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, page 14. IEEE Press, 2018.

[31] B. A. Huberman and L. A. Adamic. Internet: Growth dynamics of the World-Wide Web. *Nature*, 1999.

[32] 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.

[33] H. Jeong, B. Tombor, R. Albert, Z. N. Oltvai, and A.-L. Barabási. The large-scale organization of metabolic networks. *Nature*, 2000.

[34] Y. Ji, H. Liu, and H. H. Huang. iSpan: Parallel Identification of Strongly Connected Components with Spanning Trees. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, page 58. IEEE Press, 2018.

[35] X. Ju, D. Williams, H. Jamjoom, and K. G. Shin. Version Traveler: Fast and Memory-Efficient Version Switching in Graph Processing Systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 523–536. USENIX Association, 2016.

[36] U. Kang, H. Tong, J. Sun, C.-Y. Lin, and C. Faloutsos. GBASE: A Scalable and General Graph Management System. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2011.

[37] A. D. Kent. Comprehensive, Multi-Source Cyber-Security Events. Los Alamos National Laboratory, 2015.

[38] A. D. Kent, L. M. Liebrock, and J. C. Neil. Authentication graphs: Analyzing user behavior within an enterprise network. *Computers & Security*, 48:150–166, 2015.

[39] U. Khurana and A. Deshpande. Efficient Snapshot Retrieval over Historical Graph Data. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 997–1008. IEEE, 2013.

[40] K. Kim, T. Wang, R. Johnson, and I. Pandis. Ermia: Fast Memory-Optimized Database System for Heterogeneous Workloads. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1675–1687. ACM, 2016.

[41] D. Knoke and S. Yang. *Social network analysis*, volume 154. Sage, 2008.

[42] J. Kreps, N. Narkhede, J. Rao, et al. Kafka: a Distributed Messaging System for Log Processing. In *Proceedings of the NetDB*, pages 1–7, 2011.

[43] P. Kumar and H. H. Huang. G-Store: High-Performance Graph Store for Trillion-Edge Processing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2016.

[44] P. Kumar and H. H. Huang. Falcon: Scaling IO Performance in Multi-SSD Volumes. In *2017 USENIX Annual Technical Conference (USENIX ATC 17). USENIX Association*, pages 41–53, 2017.

[45] P. Kumar and H. H. Huang. SafeNVM: A Non-Volatile Memory Store with Thread-Level Page Protection. In *IEEE International Congress on Big Data (BigData Congress), 2017*, pages 65–72. IEEE, 2017.

[46] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *WWW*, 2010.

[47] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: Large-Scale Graph Computation on Just a PC. In *OSDI*, 2012.

[48] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graphs over time: densification laws, shrinking diameters and possible explanations. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 177–187. ACM, 2005.

[49] H. Liu and H. H. Huang. Enterprise: Breadth-First Graph Traversal on GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2015.

[50] H. Liu and H. H. Huang. Graphene: Fine-Grained IO Management for Graph Computing. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST)*, 2017.

[51] H. Liu, H. H. Huang, and Y. Hu. iBFS: Concurrent Breadth-First Search on GPUs. In *Proceedings of the SIGMOD International Conference on Management of Data*, 2016.

[52] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *Proceedings of the VLDB Endowment (VLDB)*, 2012.

[53] S. Maass, C. Min, S. Kashyap, W. Kang, M. Kumar, and T. Kim. Mosaic: Processing a Trillion-Edge Graph on a Single Machine. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, 2017.

[54] P. Macko, V. J. Marathe, D. W. Margo, and M. I. Seltzer. LLAMA: Efficient Graph Analytics Using Large Multiversioned Arrays. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*, pages 363–374. IEEE, 2015.

[55] J. Malicevic, B. Lepers, and W. Zwaenepoel. Everything you always wanted to know about multicore graph processing but were afraid to ask. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 631–643, Santa Clara, CA, 2017. USENIX Association.

[56] R. C. McColl, D. Ediger, J. Poovey, D. Campbell, and D. A. Bader. A Performance Evaluation of Open Source Graph Databases. In *Proceedings of the First Workshop on Parallel Programming for Analytics Applications*, PPAA '14, pages 11–18, New York, NY, USA, 2014. ACM.

[57] F. McSherry, M. Isard, and D. G. Murray. Scalability! But at what COST? In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, 2015.

[58] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A Timely Dataflow System. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[59] Neo4j Inc. https://neo4j.com/, 2016.

[60] D. Nguyen, A. Lenharth, and K. Pingali. A Lightweight Infrastructure for Graph Analytics. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[61] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: bringing order to the Web. 1999.

[62] 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.

[63] I. Robinson, J. Webber, and E. Eifrem. *Graph Databases*. O'Reilly Media, 2013.

[64] D. M. Romero, B. Meeder, and J. Kleinberg. Differences in the mechanics of information diffusion across topics: idioms,

political hashtags, and complex contagion on twitter. In *Proceedings of the 20th international conference on World wide web*, pages 695–704. ACM, 2011.

[65] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel. Chaos: Scale-out Graph Processing from Secondary Storage. In *SOSP*. ACM, 2015.

[66] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: Edge-centric Graph Processing using Streaming Partitions. In *SOSP*. ACM, 2013.

[67] S. Sallinen, K. Iwabuchi, S. Poudel, M. Gokhale, M. Ripeanu, and R. Pearce. Graph Colouring as a Challenge Problem for Dynamic Graph Processing on Distributed Systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 30. IEEE Press, 2016.

[68] J. Seo, J. Park, J. Shin, and M. S. Lam. Distributed SociaLite: A Datalog-Based Language for Large-Scale Graph Analysis. *Proceedings of the VLDB Endowment*, 6(14):1906–1917, 2013.

[69] M. Sevenich, S. Hong, O. van Rest, Z. Wu, J. Banerjee, and H. Chafi. Using Domain-specific Languages for Analytic Graph Databases. *Proc. VLDB Endow.*, 9(13):1257–1268, Sept. 2016.

[70] B. Shao, H. Wang, and Y. Li. Trinity: A Distributed Graph Engine on a Memory Cloud. In *Proceedings of the SIGMOD International Conference on Management of Data*, 2013.

[71] F. Sheng, Q. Cao, H. Cai, J. Yao, and C. Xie. GraPU: Accelerate Streaming Graph Analysis Through Preprocessing Buffered Updates. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, 2018.

[72] 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.

[73] J. Shun and G. E. Blelloch. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP)*, 2013.

[74] M. J. M. Turcotte, A. D. Kent, and C. Hash. Unified Host and Network Data Set. *ArXiv e-prints*, Aug. 2017.

[75] K. Vora, R. Gupta, and G. Xu. Kickstarter: Fast and Accurate Computations on Streaming Graphs via Trimmed Approximations. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 237–251. ACM, 2017.

[76] M. Wu, F. Yang, J. Xue, W. Xiao, Y. Miao, L. Wei, H. Lin, Y. Dai, and L. Zhou. GRAM: Scaling Graph Computation to the Trillions. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, 2015.

[77] W. Xie, Y. Tian, Y. Sismanis, A. Balmin, and P. J. Haas. Dynamic interaction graphs with probabilistic edge decay. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*, pages 1143–1154. IEEE, 2015.

[78] 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.

[79] D. Zheng, D. Mhembere, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay. FlashGraph: Processing Billion-Node Graphs on an Array of Commodity SSDs. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, 2015.

[80] X. Zhu, W. Chen, W. Zheng, and X. Ma. Gemini: A Computation-Centric Distributed Graph Processing System. In *OSDI*, pages 301–316, 2016.

[81] X. Zhu, W. Han, and W. Chen. GridGraph: Large-scale Graph Processing on a Single Machine Using 2-level Hierarchical Partitioning. In *Proceedings of the USENIX Conference on Usenix Annual Technical Conference*, 2015.