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

PRADEEP KUMAR, Department of Computer Science, William and Mary
H. HOWIE HUANG, Department of Electrical and Computer Engineering,
George Washington University

There is a growing need to perform a diverse set of real-time analytics (batch and stream analytics) on evolving graphs 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; rather, they excel in one area that keeps a private data store in a specialized way to favor their operations only. To address this challenge, we have designed and developed GraphOne, a graph data store that abstracts the graph data store away from the specialized systems to solve the fundamental research problems associated with the data store design. It 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 of data ingestions (called *data visibility*) for concurrent execution of diverse classes of real-time graph analytics with only a small data duplication. Experimental results show that GraphOne is able to deliver 11.40× and 5.36× average speedup in ingestion rate against LLAMA and Stinger, the two state-of-the-art dynamic graph systems, respectively. Further, they achieve an average speedup of 8.75× and 4.14× against LLAMA and 12.80× and 3.18× against Stinger for BFS and PageRank analytics (batch version), respectively. GraphOne also gains over 2,000× speedup against Kickstarter, a state-of-the-art stream analytics engine in ingesting the streaming edges and performing streaming BFS when treating first half as a base snapshot and rest as streaming edge in a synthetic graph. GraphOne also achieves an ingestion rate of two to three orders of magnitude higher than graph databases. Finally, we demonstrate that it is possible to run concurrent stream analytics from the same data store.

CCS Concepts: • **Information systems → Data management systems**; **Main memory engines**; **Information storage systems**;

Additional Key Words and Phrases: Graph systems, graph data management, unified graph data store, batch analytics, stream analytics

This article is an Extension of a FAST'19 Conference Paper by Kumar et al. [48]. There has been significant addition to the conference paper, including presentation on concurrent analytics execution, comparison against two new systems: a snapshot-based batch graph analytic system and a stream graph analytics system. We also incorporate various new results on raw text data, evaluation of overhead on GraphView, many new evaluations on design parameters, and extended discussions using new diagrams and tables to improve the readability throughout the article.

This work was supported in part by National Science Foundation CAREER award 1350766 and grants 1618706 and 1717774.

Authors' addresses: P. Kumar, Department of Computer Science, William and Mary, 251 Jamestown Road, Williamsburg, VA, 23185; email: pkumar@wm.edu; H. H. Huang, Department of Electrical and Computer Engineering, George Washington University, 800 22nd Street NW, Washington, DC, 20052; email: howie@gwu.edu.

1553-3077/2020/01-ART29 $15.00
https://doi.org/10.1145/3364180

ACM Reference format:
Pradeep Kumar and H. Howie Huang. 2020. GraphOne: A Data Store for Real-time Analytics on Evolving Graphs. *ACM Trans. Storage* 15, 4, Article 29 (January 2020), 40 pages.
https://doi.org/10.1145/3364180

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 [35], social networks [22, 43, 49], and the web [9, 33]. 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 [69], graph traversal [11, 54, 56]) that analyzes a static snapshot of the data, and *stream analytics* (e.g., anomaly detection [8], topic detection [73]), which 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 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 [10, 77, 78, 93]. The key requirement here is to sustain a large volume of fine-grained updates at a high velocity and simultaneously provide high-performance data access for various classes of real-time analytics and query support.

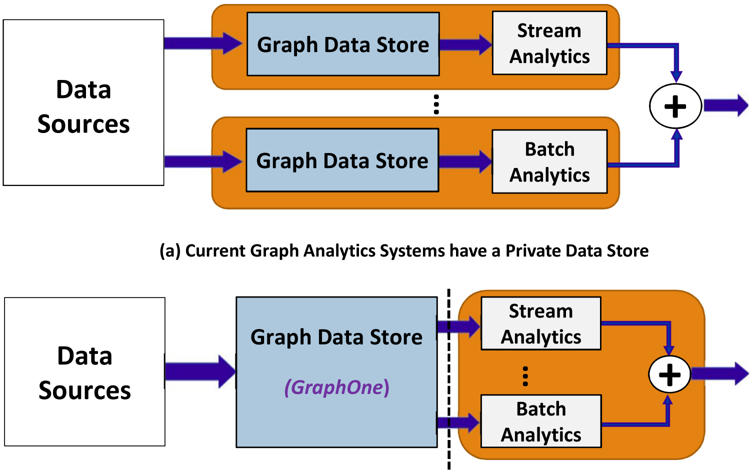
This trend poses a number of challenges to the underlying storage and data management system. First, batch and stream analytics perform different kinds of data access, that is, the former visits the whole graph, while the latter focuses on the data within a time window. Second, each analytic has a different notion of real time, that is, data are 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 provide neither diverse data access nor at different granularity of data ingestion in the presence of a high data arrival rate, as shown in Table 1. Many dynamic graph systems [50, 59] only support batched updates, and a few others [23, 79] offer data visibility at fine granularity of updates but with a weak consistency guarantee (atomic read consistency), 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 [66] can handle fine-grained updates but suffer from poor ingestion rate for the sake of strong consistency guarantee [62]. Also, such systems are not designed to support high-performance streaming data access over a time window. However, graph stream engines [34, 65, 76, 85] 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 as shown in Figure 1(a). However, such an approach would be suboptimal [93], 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.

Table 1. A Simplified Classification of Various Graph Data Management Systems Depicting That Each Graph System Only Supports One Type of Data Ingestion and Access Pattern

Graph Systems	**Ingestion Type Offered**	**Data Access Type Offered**
Graph Database	Fine-grained (e.g., Neo4j [66])	Whole Data
Dynamic Graph System	Fine-grained (e.g., Stinger [23])	Whole Data
	Coarse-grained (e.g., Graphchi [50])	
Stream Graph System	Coarse-grained (e.g., Naiad [65])	Streaming Access of Whole Data
	Coarse-grained (e.g., GraphTau [34])	Streaming Access from Time-window



(a) Current Graph Analytics Systems have a Private Data Store

(b) GraphOne abstract Data Store away from Graph Analytics Systems

Fig. 1. Showing a high-level overview of graph analytics system. The evolving graph data arrive one edge at a time or in batches from the data sources, such as Apache Kafka, database change log, and so on. Each of current graph analytics systems have a private data store optimized for one class of usecases that interface with data sources. GraphOne abstracts the data store away from the analytics systems to solve the problems associated with a general-purpose graph data store and enables execution of diverse classes of analytics from the same data store using GraphView instances.

In this work, we have designed GraphOne, a unified graph data store abstracted away from the specialized graph systems, as shown in Figure 1(b), to solve the fundamental problem of such data-store design and management. GraphOne offers diverse data access at two granularity levels of data ingestions for various real-time analytics, so that many classes of analytics can be performed from the same data store, while supporting data ingestion at a high arrival rate. Figure 2 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 using a logging phase 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 using a archiving phase and is optimized for batch and streaming analytics. It is important to note that the graph data are not duplicated in two formats, although a small amount of overlapping is allowed to keep the original composition of the versions intact.

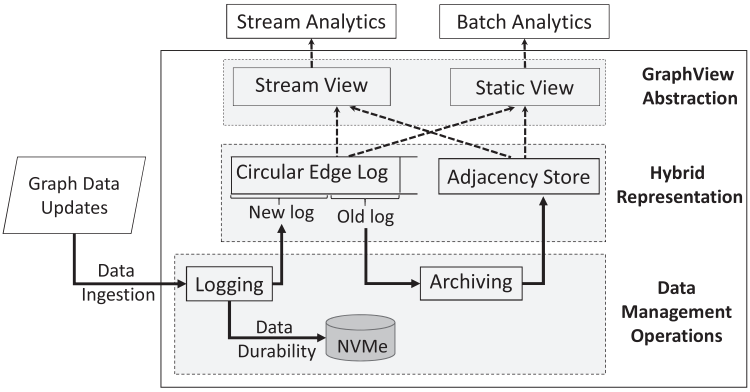


Fig. 2. High-level architecture of GraphOne. Solid and dotted arrows show the data management and access flow, respectively.

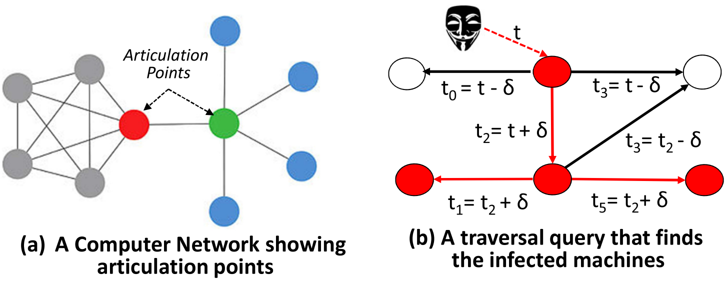
GraphOne enforces data ordering using the temporal nature of the edge log in the logging phase and keeps the per-vertex edge arrival order intact in the adjacency store using a novel *edge sharding* in the archiving phase. A *dual versioning* technique then exploits the fine-grained versioning of the edge list format and the coarse-grained versioning of the adjacency list format to create real-time versions to offer *snapshot isolation* data consistency. 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 in case of stream analytics. This is different from the current stream graph systems where the size of batched update is often determined by the heaviest compute, which hurts the timeliness of other concurrently executing analytics.

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 Figure 2: (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 choose 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 (Section 7.4).

We have implemented GraphOne as an in-memory graph data store with a limited durability guarantee on external non-volatile memory express solid-state drives (NVMe SSD). For comparison, we have evaluated it against many classes of in-memory graph systems: LLAMA [59], as a snapshot-based coarse-grained graph batch analytics system; Stinger [23], a fine-grained dynamic graph system; Neo4j and SQLite, two graph data management systems; and Kickstarter [85], as graph stream analytics. The experimental results show that GraphOne can support a high data ingestion rate,; specifically, it achieves 11.40× average speedup in ingestion rate than LLAMA, 5.36× higher ingestion rate than Stinger, and two to three orders of magnitude higher ingestion rate than graph databases.

In addition, GraphOne outperforms LLAMA by 8.75× and 4.14× and Stinger by 12.80× and 3.18× for BFS and PageRank analytics (batch version), respectively. GraphOne also gains a



(a) A Computer Network showing articulation points

(b) A traversal query that finds the infected machines

Fig. 3. Graph traversal can locate possible infected nodes using real-time authentication graph if infected user and node are known.

speedup of over 2,000× over Kickstarter in ingesting the streaming edges and performing streaming BFS when treating the first half as a base snapshot and the rest as streaming edge in a synthetic graph. The stream processing in GraphOne runs parallel to data archiving phase (update to adjacency store) that offers 28.58% higher ingestion rate compared to the current practice of running the two in sequence. Due to this feature, GraphOne also enables concurrent stream analytics from the same dynamic adjacency store that is mostly absent in prior graph stream analytics engine.

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 for concurrent execution of various usecases with GraphView and data visibility abstractions.

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

2 USE CASE: NETWORK ANALYSIS

Graph analytics is a natural choice for data analysis on an enterprise network. Figure 3(a) shows a graph representation of a simple computer network. Such a network can be analyzed in its entirety by calculating the diameter [52] 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 dataflow 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 [18, 40].

The Los Alamos National Laboratory (LANL) recently released a comprehensive dataset [39] that captures a wide range of network information, including authentication events, process events, DNS lookups, and network flows. The LANL data cover over 1.5 billion events, 12,000 users, and 17,000 computers and span 58 consecutive days. For example, the network authentication data capture the login information with which a user logs into a network machine and 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 traversal query on the real-time authentication graph

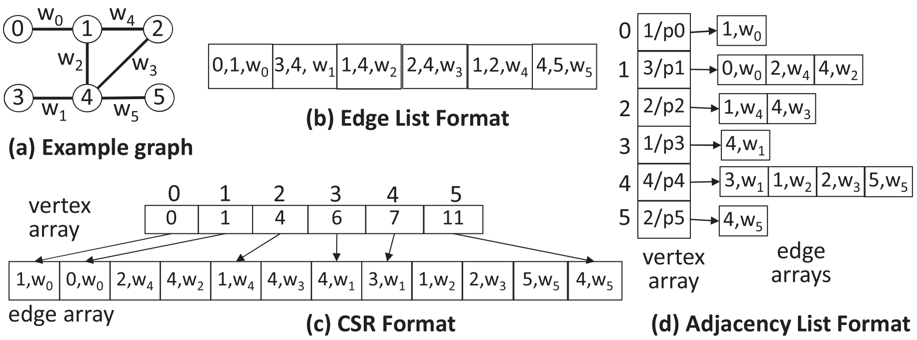

0
1
2
3
4
5
w_0
w_4
w_2
w_3
w_1
w_5
(a) Example graph
0,1,w_0 | 3,4, w_1 | 1,4,w_2 | 2,4,w_3 | 1,2,w_4 | 4,5,w_5
(b) Edge List Format
vertex array
0 1 2 3 4 5
0 | 1 | 4 | 6 | 7 | 11
1,w_0 | 0,w_0 | 2,w_4 | 4,w_2 | 1,w_4 | 4,w_3 | 4,w_1 | 3,w_1 | 1,w_2 | 2,w_3 | 5,w_5 | 4,w_5
edge array
(c) CSR Format
0 1/p0 → 1,w_0
1 3/p1 → 0,w_0 | 2,w_4 | 4,w_2
2 2/p2 → 1,w_4 | 4,w_3
3 1/p3 → 4,w_1
4 4/p4 → 3,w_1 | 1,w_2 | 2,w_3 | 5,w_5
5 2/p5 → 4,w_5
vertex array
edge arrays
(d) Adjacency List Format

Fig. 4. Sample graph and its various storage format. GraphOne concentrates on the complementary properties of the edge list format and the adjacency list format.

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 [40], as shown in Figure 3(b).

In summary, a high-performance graph store that captures dynamic data in the network, combined with user and machine information and network topology is advantageous in understanding the health of the network, accelerating network service, and protecting it against various attacks. This clearly articulates the need to have a data store that can efficiently ingest the incoming data and make it available for analysis in a form that various analytics and query engine prefer, so that analytics writer can focus on the actual usecases rather than thinking about how to manage the incoming data and providing that right data consistency as well proving various data access types. To this end, GraphOne presents a graph data storage and management infrastructure that abstracts the data store away from the analytics engines to solve the basic data access problem associated with such diverse real-time analytics.

3 OPPORTUNITIES AND OVERVIEW

A graph can be defined as G = (V, E, W), where V is the vertex set, 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

Figure 4 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, a *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, so this is not the optimal choice for many graph queries and batch analytics, wher it is preferred to get the neighboring edges of a vertex quickly [12, 31, 32, 36], and so on. However, the edge list provides a natural support for fast update, as each update simply needs to be appended to the end of the list. However, the adjacency list format loses the temporal ordering, as the incoming updates get scattered over the edge arrays; thus, this is not suited directly for stream analytics. Also, creation of fine-grained snapshots are not easy on adjacency lists, hence prior work has only attempted to

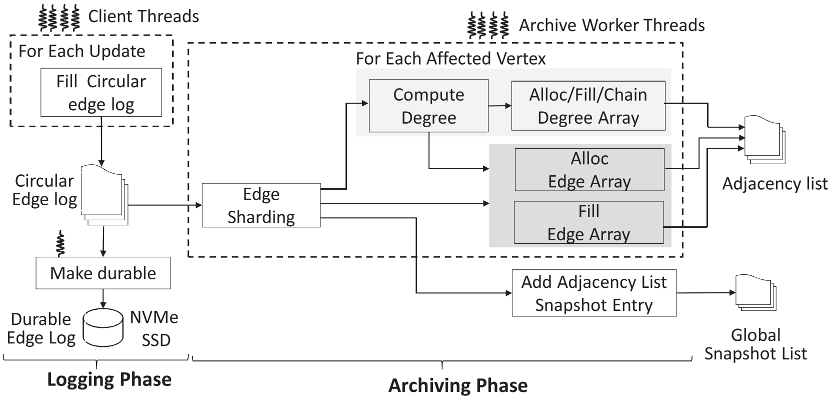


Fig. 5. Architecture of GraphOne. Operations related to same data structures have been grayed out in the archiving phase. The compaction Phase is not shown.

create coarse-grained versioning [28, 59]. Stinger [23] directly updates the adjacency list format but can provide read committed only, a weaker data consistency, that does not provide any snapshot capability. Therefore different steps within an analytics will run on different versions of data, thereby producing unknown results in the case of evolving graphs.

Given their advantages and disadvantages, neither format is ideally suited for supporting both the batch and stream analytics and high ingestion of data 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 that requires sequential write only. However, 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.

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. Hence, the adjacency list format can use the fine-grained snapshot capability of the edge list to complement its coarse-grained snapshot capability [28, 59] to create real-time versions of graph data.

3.2 Overview

GraphOne utilizes a hybrid graph data store (discussed in Section 4) that consists of a small circular *edge log* and the *adjacency store*. Figure 5 shows a high-level overview of GraphOne architecture. The hybrid store is managed in several phases (presented in Section 5). Specifically, during the *logging phase*, the edge log records the incoming updates in the edge list format atomically 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 in an asynchronous manner and requires support from upstream buffering for complete recovery. In this article, we use *data ingestion* to mean that newly arrived edges has been percolated to the adjacency store.

Thus, GraphOne is an in-memory graph data store with a weaker durability guarantee and offers snapshot isolation as its data consistency, which is slightly weaker than full serializability that supports transactions, but is stronger than the read-committed isolation level offered by NoSQL data stores such as Stinger [23] or Trinity [79, 90]. Due to the logging phase, the non-archived edges move to the adjacency store in batches, which enables a number of optimizations to improve the performance of archiving phase (Section 5.1.2) and to also reduce the memory consumption of the adjacency store (Section 5.2). These techniques together bring the performance of GraphOne very close to static graph engines.

To efficiently create and manage immutable graph versions for real-time data analytics in the presence of the incoming updates, we provide a set of GraphView APIs (discussed in Section 6) that data access for diverse classes of analytics. Specifically, *static view* API is for batch processing, while *stream view* API is for stream processing. Internally, the views utilize a *dual versioning* technique where the versioning capability of both formats is exploited for faster creation of a snapshot of the current data at very low cost. For example, a real-time static view can readily be composed by using the latest coarse-grained version of the adjacency store and the latest fine-grained version of non-archived edges. The *data overlap* technique allows GraphOne to keep the original composition of the created view intact and thus guarantees its data access without interfering the data management operations.

It is important to note that the GraphView also provides analytics with the flexibility to trade off the granularity of *data visibility* of incoming updates for better analytics performance, i.e., an analytic gets to choose the type of data ingestion that it is interested in. For example, the analytics that prefer running only on batched updates will access the latest adjacency store and thus avoid the cost associated with the access of the latest edges from the non-archived edges that are not indexed by source vertices.

Despite its simplicity, the abstraction of *data visibility* is very powerful, as it allows high-performance fine-grained ingestion but offers the performance of batched updates to those analytics that are not interested in fine-grained ingestions. Put other way, GraphOne removes the cost of fine-grained ingestion from the data write path to the data read path of those analytics that are really interested in running on fine-grained ingestion. All other data read paths offer performance of a batched update system.

3.3 Major Differences

GraphOne is different than databases, graph batch analytics, graph stream analytics, and key-value stores in some major ways. In this sub-section, we provide an overview of those differences.

In databases, it is possible to provide functionality similar to adjacency list graph format. For example, a database can create a simple schema to store graph edges containing source and destination vertices in an edge table. The data of edge table is similar to an edge list format. Creating an index in the source vertex in a edge table results in a structure similar to an adjacency list. In that case, databases update the edge list log structure (for logging purpose) and update to the indexed structures (archiving to adjacency store) in the same thread context. Stinger [23] does not have any log structure and maintains an indexed structure only but works same way, i.e., update of the indexed structure happens in the thread context of update thread.

However, GraphOne updates both of them in separate thread contexts, while delaying the update of the indexed structure for batching the edges to achieve better overall ingestion throughput. The major difference of GraphOne from the snapshot-based graph batch analytics on evolving graphs, such as Kineograph [17], LLAMA [59] and so on, is the ability of GraphOne to create fine-grained snapshots and control the data access of those snapshots by anlaytics using the data visibility abstraction.

Both the databases and snapshot-based graph batch analytics and GraphOne run analytics in separate thread contexts than the data ingestion.

For graph stream analytics, update of the log structure happens separately than the analytics in GraphOne and prior stream engines. The major difference is in the context of the indexed structure update (archiving phase). Prior graph stream engines perform the archiving phase in the context of stream analytics using operators such map, join, and group-by (as in Naiad [64], GraphTau [34], etc.) or using specific implementation (as in Kickstarter [85], GraphBolt [61], etc.), followed by actual stream computation. GraphOne, however, moves the archiving phase out of the analytics pipeline, i.e., archiving happens in parallel to actual stream computation, which results in overall better performance.

Due to the overlapping nature of archiving and computation, GraphOne becomes the first-ever system that can enable concurrent stream analytics from the same indexed structure. Even snapshot-based stream analytics systems such as Kinerograph [17] cannot execute two classes of stream analytics together: Those that need to access whole data, and those that require window access, as Kineograph, propose a different snapshot technique to be implemented as future work for data-window access (see Section 6.3 in the article [17]). Prior stream analytics can decides to deploy multiple instances of same stream analytics engine to run concurrent stream analytics but will also produce a separate adjacency store.

Key-value store natively provides only three basic operations: PUT, GET, and DELETE. However, the APPEND operation is a must-have requirement for graph systems to support graph data ingestion. A new edge insertion potentially means appending a new entry in the neighbor list (the value) of the source vertex (the key). Similarly, a deletion of edge requires deleting one entry from neighbor list of the source vertex. Many prior graph analytics systems, such as Kineograph, Trinity, and so on, additionally implement a graph layer on top of a key-value store to support this APPEND operation. The append operation on top of key-value store works using copy-on-write in Trinity: An edge addition requires copy of the prior neighbor list (value) of the source vertex (key) from the memory and then appends a new neighbor to the neighbor list and, finally, writes the whole neighbor list to a new memory location. Then, they provide some optimizations to reduce this migration on each append.

Clearly, key-value store is not natively suited for storing evolving graph data. However, GraphOne provides a native implementation of the adjacency list with in-built support for the APPEND operation without using copy-on-write approach.

Finally, the GraphOne data store brings all different classes of analytics within a single system by allowing them to be performed concurrently from the same data store using the GraphView interface. In this article, we concentrate on the data-store design and optimization and demonstrate through discussions and experiments that its data-store abstraction and optimization techniques can be integrated to many classes of analytics engines to bring performance improvement. For this, we will be implementing actual analytics as suggested in prior works as much as possible for fair comparison while doing data-management our way.

4 HYBRID STORE AND VERSIONING

In this section, we first present internals of hybrid store followed by a discussion on versioning.

4.1 Hybrid Store

The hybrid store design presented in Figure 6 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

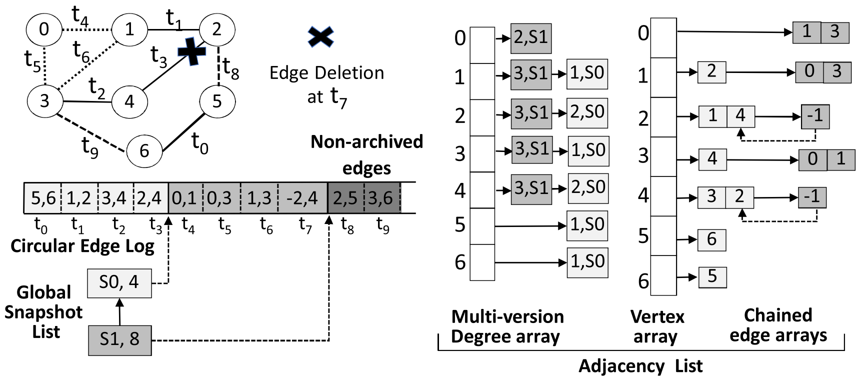


Fig. 6. The hybrid store for the data arrived from time t_0 to t_9: The vertex array contains pointers to the first and the last block of each edge array, while degree array contains deleted and added edge counts. However, only the pointer to the first block in the vertex array, and total count in the degree array are shown for brevity.

the edge log 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 Figure 6 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 a *vertex array*, per-vertex *edge arrays*, and a multi-versioned *degree array*. The *vertex array* contains a per-vertex flag and pointers to the first and last blocks 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 arrays* 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 that is linked to the last block. Figure 6 shows chained edge arrays for the vertices with ID 1 to 4 for data updates that arrive 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 are 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 and is the most important data structure to support multi-versioning of adjacency list, because 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. We exploit this property and customize the degree structure to provide snapshot capability even in presence of data deletion. 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 3). 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 Figure 6 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

Figure 6. The degree array nodes of an older versions are 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 IDs 1–4 can be reused by later snapshots (e.g., S2). The garbage collection of degree node is part of archiving phase (see Section 5.1.2).

The *vertex array* is separated from the degree array for easier implementation of multi-versioning of the adjacency list, as the degree array nodes corresponding to older snapshots will be freed and recycled when the corresponding snapshot retires without affecting other data structures, such as edge arrays that are actively accessed by analytics.

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. However, if edge weights are dynamic, such as network dataflow, then such weight changes are suited for various steam 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. Thus, a streaming graph will not use the tombstones that we discussed in the beginning of this sub-section.

An Example of Dataflow. The dataflow of GraphOne is presented in Figure 6. There are addition and deletion of 10 edges from time t_0 to t_9, and two snapshots of the adjacency list are created. From time t_0 to t_3, all the incoming edges are logged in the edge log. At t_4 archiving starts moving these edge log data to the adjacency list in parallel to logging. During t_4 to t_7, new edges are logged in the edge log. GraphOne again starts moving the last four updates to the adjacency list, while new arrivals are getting logged in the edge log from t_8 onward. At the end of t_9, the edge log contains two edges that are yet to move to the adjacency store.

4.2 Dual Versioning and Data Overlap

GraphOne uses *dual versioning* to create the instantaneous read-only graph views (snapshot isolation) for data analytics by exploiting both the *fine-grained* versioning property of the edge log and the *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 Figure 6, the static view at time t_6 would be an adjacency list snapshot $S0$ plus the edges from t_4 to t_6.

A small amount of *data overlap* between the two stores keeps the composition of the view intact and makes the view accessible even when the edge log data are 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.

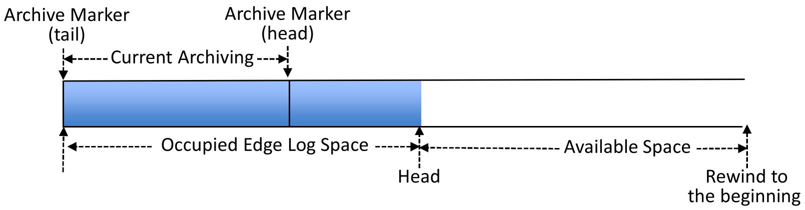


Fig. 7. Circular Edge log design showing various offset or markers. Markers for durable phase are similar to archiving and are omitted.

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 an atomic update.

5.1 Data Management Phases

Figure 5 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; this is 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 the *archiving threshold*. The logging phase continues as usual in parallel to them. Also, the archive thread and durable thread check whether 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 Figure 7. For archiving, GraphOne manages a pair of markers, i.e., the archiving operation happens from 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, if required, and acquires an edge list format. The mapping between vertex label to vertex ID is managed using a hashmap. The reverse mapping to map the vertex ID to vertex name is managed internally. Many times, the incoming data may directly contain a unique pair of vertex ID in an edge. For such cases, user can configure GraphOne to directly use the existing ID space. In either case, 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. The source vertex also stores the operator (Section 4), addition or deletion, along with the edges.

The atomicity of updates is ensured by the atomic increment of the head and by writing the least significant bit of the edge log rewind count in the most significant bit of the destination vertex ID. This is written after writing the source vertex and alternates between zero and 1 each time the edge log rewinds. Thus, a reader of the edge log can figure out if it is reading the latest data, old data, or a partial update. Thus, for each edge in the edge log, the MSB of source vertex contains the deletion information while the MSB of the destination vertex contains the rewind information and is written in this order.

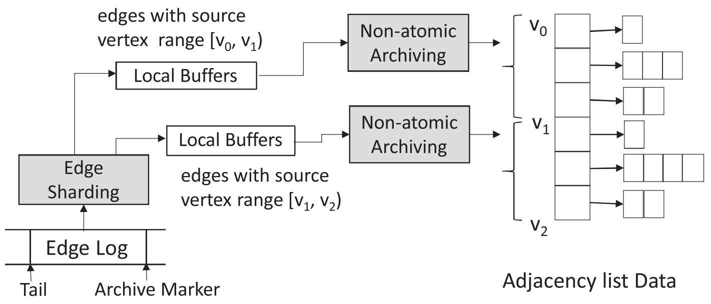


Fig. 8. Edge sharding separates the non-archived edges into many buffers based on their source vertex ID, so that the per-vertex edge arrays can keep the edge log arrival order and enable non-atomic archiving.

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. This phase has many stages and is discussed next.

Edge Sharding. 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, then the edge may become alive or dead in the edge arrays depending on the archiving order of the two data points. To avoid this ordering problem, we propose an *edge sharding* technique.

An *edge sharding* stage in the archiving phase (Figure 8) 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, i.e., each local buffer contains edges belonging to a few particular source vertices. Our implementation makes sure that the per-vertex edges in the local buffers are in the same arrival order as that of the edge log. This is implemented by scanning the edge log twice. In the first scan, each worker thread counts its contribution to all the local edge buffers, and in the second scan, the worker thread copies the edges to the corresponding slots in those local buffers.

For undirected graphs, the total edge count in the local buffer is twice the non-archived edge count, as the ordering of reverse edges is also managed. For directed edges, both directions have their own local buffers. Hence, each edge is stored twice, one in the forward direction and another in the reverse direction. However, GraphOne provides a way to avoid storing the edges twice, which is called a *unigraph* in this article.

Non-Atomic Archiving. Edge sharding has an additional advantage of avoiding the usage of atomic instructions for populating the edge arrays, i.e., the edges in each local buffer are 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 a 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 [42], which 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. If a previous edge array for a participating vertex exists, then the new edge block is chained to it; otherwise, a new entry is created for the new edge block in the vertex array. 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 are 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 a 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 [44], 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 [47].

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, as it will break access of whole data in case of batch analytics or full access by streaming analytics. During checkpointing the adjacency store, the vertex ID, and length of the edge array are maintained 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 Section 5.1.2. The compaction needs a similar reference counting for the private static views (Section 6.1). For each vertex, it allocates new edge array block, 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 the 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 that move quickly to the base store, e.g, the archiving threshold of 2^{16} edges translates to only 1 MB 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 (Section 5.1.2). This leaves us with memory analysis of edge arrays, which may consume a lot of memory due to

Table 2. Impact of Two Optimizations on the Chain Count and Memory Consumption of the Edge Arrays of the Adjacency Store on the Kronecker (Kron-28) Graph

Optimizations	Chain Count		Memory
	Average	Maximum	Needed (GB)
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

excessive chaining in their edge blocks. For example, GraphOne runs an 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 148.73 GB memory and have average 29.18 chain per-vertex. We will discuss the graph datasets used in this article shortly. If all the edges were to be ingested in one archiving phase, then this static system needs only an average 0.45 chain and 33.80 GB memory. The chain count is less than 1 as 55% of 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 majority 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, as 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 2. The baseline system uses a dynamic block size that 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 [24]. They are very common in real-life graphs, such as for the twitter follower graph whose degree distribution is shown in Figure 9. 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 4 KB 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 reduced to 2.47, leading to a further reduction in memory utilization by 1.63 GB, as listed in Table 2. The reduction in memory overhead due to hub vertex handling are modest on top of cacheline-sized memory allocations. This is because the number of hub vertices are very small in comparison to total vertex of the graph. Therefore, even if one varies the threshold to identify a hub vertex, the performance will remain similar to the cacheline-sized memory (Figure 18).

6 GRAPHVIEW AND DATA VISIBILITY

GraphView data abstraction hides the complexity of the hybrid store by embedding the *data visibility* option and simplifying data access by providing APIs as shown in Tables 3, 4, and 5. The

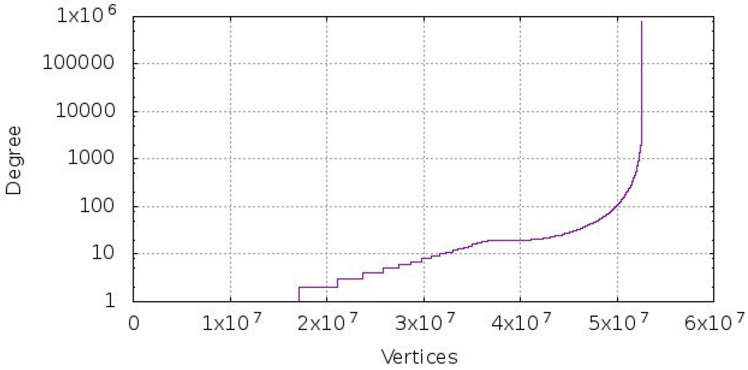


Fig. 9. Degree distribution of Twitter graph.

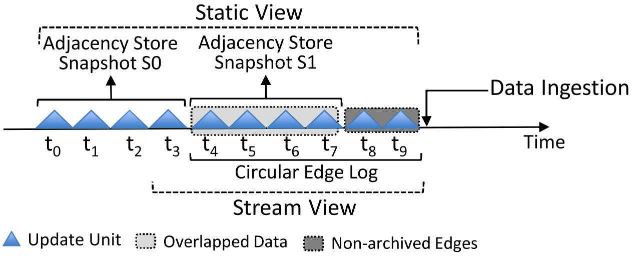


Fig. 10. GraphOne hybrid store illustrating various views with two adjacency store versions, S0 and S1, with a small edge log.

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. The access of non-archived edges provides data visibility to analytics at the edge-level granularity.

A number of views may co-exist at any time without incurring much memory overhead, as the view data are composed of the same adjacency store and non-archived edges as shown in Figure 10. Each analytics has its own view instance and executes in separate thread contexts and in parallel to any data-management phases and their threads. Generally, the registration of the batch or stream analytics triggers a thread creation that calls a function pointer specified in the create-* or reg-* APIs (Tables 3, 4, and 5), and passing the handle. The function pointer is omitted for brevity from the API signature. Caller is responsible for all the computation inside the callback, including creation of more threads to achieve parallelism if required, and a return assumes that computation has been finished. In this section, we concentrate on data-access APIs and leave the development or integration of a full-fledged analytics system to simplify some of the tasks of actual analytics as a future work.

Due to the cost of indexing the non-archived edges, GraphView provides an option to trade off the granularity of data visibility to gain analytics performance by opting to compute *only* on the latest adjacency store snapshot. Further, one can use a vertex-centric compute model [82] on the adjacency list plus an edge-centric compute model [45, 75, 96] on non-archived edges, so there is no need to index the latter as plotted later to find its optimal minimum size (Figure 16).

6.1 Static View

Batch analytics and queries prefer snapshots for computation, which can be created in real time using *create-static-view()* API, and then data may be accessed using other sets of APIs as shown in

Table 3. 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)

Table 3. A static view 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()*; otherwise, there would be a memory leak as each static view may contain at least a degree array. However, the handle can be wrapped in a reference counting objects for auto deletion when the handle goes out of scope. Based on the input supplied to *create-static-view()* API, many types of static view are defined.

ALGORITHM 1: Traditional BFS using static view APIs

```
handle ← create-static-view(global-data, private=true, simple=true)
level = 1; active-vertex = 1; status-array[root-vertex] = level;
while active-vertex do
    active-vertex = 0;
    for vertex-type v = 0; v < vertex-count; v++ do                ▷ Can be parallelized
        if status-array[v] == level then
            degree ← get-nebrs-out(handle, v, nebr-list);
            for j=0; j < degree; j++ do
                w ← nebr-list[j];
                if status-array[w] == 0 then
                    status-array[w] ← level + 1; ++active-vertex;
                end if
            end for
        end if
    end for
    ++level;
end while
delete-static-view(handle)
```

6.1.1 Basic Static View. This is the most fundamental view of data, where the data are accessed directly from underlying storage. The main low-level API are follows: *get-nebrs-archived-*()*, which returns the reference to the per-vertex edge array, and *get-non-archived-edges()*, which returns the non-archived edges. These two APIs are very useful for advanced users and for higher-level library development that requires more control and performance.

However, 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 provides simplicity to analytics writers as they do not have to know the exact internal structures of the underlying data of GraphOne. However, there is a performance cost associated with usage of this API, as it will have to repeatedly scan the non-archived edges for finding the neighboring edges of each vertex and an additional step of data copying to user-supplied buffer. This API will 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.

One can also pass *simple=true* in the *create-static-view()* to create a temporary in-memory adjacency list from the non-archived edges for optimizing the performance of get-nebr-*() API. Algorithm 1 shows a simplified BFS (push model) implementation using the high-level APIs of basic static view.

6.1.2 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, and (2) the degree array cannot be reused in 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.

The private static view approach of batch analytics is more flexible than static analytics engine where latter converts the whole data that takes much longer to complete [60] than the analytics runtime. In contrast, a private view only needs to copy a degree array whose cost will be much lower. This view is also better than other dynamic graph system that disallows the users 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.

6.1.3 Stale Static View. Many analytics are fine with data visibility at coarse-grained ingestions, and thus some stale but consistent view of the data may be better for their performance. In this case, passing *stale=true* to *create-static-view()* API returns a handle corresponding to the snapshot of the latest adjacency list only. Thus, same set of APIs can be used by users to access the slightly stale data, but performance will automatically be of batched update case. This view can be combined with private static view where degree array will be copied.

6.2 Stream View

As we discussed in the beginning of this section, all stream computations runs in a separate thread contexts, i.e., one thread per stream analytics (can be parallelized to execute the core logic of stream analytics), while logging and archiving keep working in parallel in their own thread contexts. The stream view APIs as listed in Table 4 and 5 simplify the data access in the presence of data ingestion. For a stream analytics, *window-sz* is the size on which a stream analytics want the data-state (adjacency store) to be accessible for computation and can stretch up to the beginning of the stream

Table 4. 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)

ALGORITHM 2: A stateless stream compute skeleton

```
handle ← reg-stream-view(global-data, batch-sz=10s)
init-stream-compute(handle)                              ▷ Application specific
while true do                          ▷ Or application specific exit criteria
    if update-stream-view(handle) then
        count = get-new-edges(handle, new-edges)
        for j=0; j < count; j++ do                       ▷ Can be parallelized
            do-stream-compute(handle, new-edges[j])           ▷ Or any method
        end for
    end if
end while
unreg-sstream-view(handle)
```

if not specified. The *batch-sz* means stream analytics wants to perform incremental computation only if batch-sz count of edges have been accumulated since last incremental compute.

All stream analytics in GraphOne follow a pull method, i.e., the analytics *pulls* a new snapshot by calling *update-*-view()* API at the end of incremental compute to perform the next phase of incremental compute. If no new data arrive from last time the API was called or if the size of new data is less than *batch-sz* (reg-*-view() API), then this API will get blocked and will return only when new data equal to or greater than batch-sz is available to continue the computation. Although, the focus of this sub-section is on data access for stream computations, one can pass the appropriate flag value to *reg-∗* APIs listed in Table 4 and 5 to configure the data visibility of the stream view.

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 onward in case of a fault. We now present data access patterns and corresponding APIs for two classes of stream analytics: *stateless* and *stateful* stream computations.

6.2.1 Stateless Stream Processing. A stateless computation, e.g., counting incoming edges (aggregation), only needs a batch of new edges. The APIs are listed in Table 4. The computation 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 are fetched from the durable edge log and show

Table 5. 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)

ALGORITHM 3: A stateful stream compute (vertex-centric) skeleton

```
handle ← reg-sstream-view(global-data, v-centric, stale=true)
init-sstream-compute(handle)                          ▷ Application specific
while true do                                         ▷ Or application specific exit criteria
    if update-sstream-view(handle) then
        for v=0; v < vertex-count; v++ do             ▷ Can be parallelized
            if has-vertex-changed(handle, v) then
                do-sstream-compute(handle, v)         ▷ Application specific
            end if
        end for
    end if
end while
unreg-sstream-view(handle)
```

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.

6.2.2 *Stateful Stream Processing.*

A complex stream computation, such as graph coloring [76], is stateful when it needs the streaming data and complete base store to access the computational state of the neighbors of each vertex. A variant of the static view is better suited for it, because its per-vertex neighbor information eases the access of the computational state of neighbors. The APIs are listed in Table 5. The stateful computation 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 identified using the *has-vertex-changed()* API. This is updated during teh *update-sstream-view()* call, which also updates the degree array. Algorithm 3 shows an example code snippet.

As the degree array plays an important role in 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 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.

Table 6. Historic View APIs

count	get-prior-edges(global-data, start, end, ptr)

6.2.3 Discussion. In this sub-section, we bring the differences to the existing graph stream analytics. The major difference arises for stateful computation, as computation requires the access of prior data in the form of an adjacency store that is built in separate thread context. While GraphOne performs archiving in a separate thread context than that of the analytics, a prior steam analytics engine performs both of them in sequence in the same thread context. This difference can easily be explained using Algorithm 3. Prior stream engines will perform an actual archiving phase (building/updating adjacency store indexes) at line number 4 instead of calling *update-sstream-view(handle)*, while GraphOne does a very lightweight metadata update at line number 4 by calling the above-mentioned function so that the view handle can point to the latest adjacency store as the update to the adjacency store happens in a different thread context. The rest of the lines in Algorithm 3 remain the same. Thus, it will be possible to integrate many stream analytics systems with a GraphOne data store. This difference also enables concurrent execution of multiple stream analytics from the same adjacency store.

For stateless computation, there is no difference with the existing stream engines, as analytics do not require access to the adjacency store at all. If these are the only analytics running in the system, then GraphOne can be configured to not perform the archiving phase. Thus, prior streams systems will be identical to GraphOne, and both of them look exactly like Algorithm 2.

6.3 Historic Views

GraphOne provides many views from the recent past, but it is not designed for getting arbitrary historic views from the adjacency store. However, a durable edge log can provide the same using a *get-prior-edges()* API (Table 6) in edge list format as it keeps deleted data, behaving similarly to existing data stores [14, 25].

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 resultant 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 degree arrays at both the offsets of the time-window of interest 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 and 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 two Intel Xeon CPU E5-2683 sockets, each having 14 cores with hyper-threading enabled. It has 512-GB 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

Table 7. Graph Datasets Showing Vertex and Edge Counts in Millions (for Deletions See Section 7.3)

Graph Name	Graph Type	Vertex Counts (in Millions)	Edge Counts (in Millions)
LANL	Directed	0.16	1,521.19
Twitter	Directed	52.58	1,963.26
Friendster	Directed	68.35	2,586.15
Subdomain	Directed	101.72	2,043.20
Kron-28	Undirected	256	4,096
Kron-21	Undirected	2	32

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, and so on.

Datasets. Table 7 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 [27], all with 4-byte vertex size and without any weights. We also use these graphs for running comparisons with LLAMA, which always expects weighted graphs. So these graphs are also converted to a weighted graph with 4 bytes of random weights assigned by a tool developed as part of the LLAMA software suite. The LANL network flow dataset [84] 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 an experiment on the first 10 days of data. We test deletions on a weighted RMAT graph [15] generated by a graph generator developed by McColl et al. [62] where vertex and weight sizes are 8 bytes. It contains 4 million vertices, 64 million edges, and a update file containing 40 million edges, from which 2,501,937 edges are for deletions.

These graphs have been taken from various domains, such as Twitter and Friendster from social media, a subdomain from a web domain, while LANL is captured from an enterprise cyber network apart from many synthetic graphs. We believe these are diverse enough to capture a wide variety of applications. We have two copies of each dataset, one in binary edge list format and another in raw text edge list format. During the ingestion, vertex name–to–vertex ID conversion was not needed for binary graph data, as we directly used the vertex ID supplied with these datasets. This convention is also followed by other graph systems. We discuss two cases of raw text data in Section 7.2. 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.

7.1 Data Ingestion Performance on Binary Data

In this sub-section, we present data ingestion when they already contain the data in binary edge format, i.e., GraphOne does not create any vertex ID from the incoming data. This will depend on the data source; however, this is a perfect scenario to test the performance of various stages of data management pipeline and the end-to-end ingestion rate as data are already in a ready state to be ingested in GraphOne.

Logging and Archiving Rate. Logging to edge log is naturally faster, while archiving rate depends upon the archiving threshold. Table 8 lists the logging rate of a thread and the archiving rate at the archiving threshold of 2^{16} edges for our graph dataset. A thread can log close to

Table 8. Different Data Management Rates in Millions Edges/s (M/s) on Various Graph Dataset

Graph	Individual Phases (M/s)		In-Memory Rate (M/s)		Ext-Memory Rate (M/s)		Compaction
Name	Logging	Archiving	Ingestion	Recovery	Ingestion	Recovery	Rate (M)
LANL	35.98	28.91	26.99	30.23	25.26	29.48	41.85
Twitter	82.62	47.98	66.39	71.28	61.13	71.87	541.71
Friendster	82.85	49.32	60.40	95.78	58.35	95.44	520.65
Subdomain	82.86	43.43	68.25	180.75	61.54	151.96	444.84
Kron-28	79.23	43.68	52.39	116.18	49.70	107.61	798.91
Kron-21	78.91	78.40	58.31	90.44	57.02	66.66	1011.68

The results show that the ingestion rate would be upper and lower bounded by the logging and archiving rate. See Section 7.3 for results on datasets containing deletions.

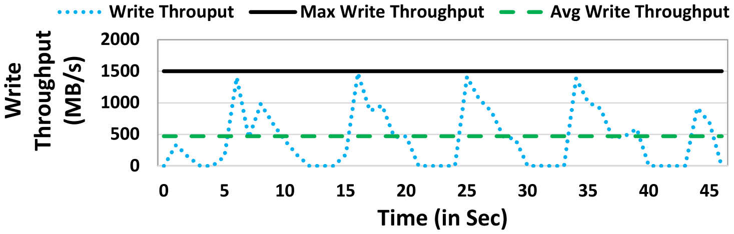


Fig. 11. Write throughput for friendster in GraphOne comparing against average requirement and maximum available in an NVMe.

80 million edges per second, while the archiving rate is only around 45 million edges per 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 maintained 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 the archiving rate (at the archiving threshold) except in Kron-21, as edges more than the archiving threshold are archived in each epoch due to a higher logging rate. This indicates that GraphOne can support a higher arrival rate as the 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 have 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 of a total 104 million edges. Results for other graphs are shown in Table 8. The poor rate for the LANL graph is due to the long tail for compacting edge arrays of few vertices. As shown in Figure 19, the compaction improves the analytics performance where the static GraphOne serves a compacted adjacency list as it had no link in its edge arrays.

Durability. The durable phase has less than 10% impact on the ingestion rate. Table 8 shows the in-memory ingestion rate and can be compared against that of GraphOne, which uses NVMe SSD for durability. This is because the durable phase runs in a separate thread context and exhibits only sequential write. The NVMe SSD can support up to 1,500 MB/s sequential write, and that is sufficient for GraphOne, as it only needs smaller write IO throughput, as shown in Figure 11 for Friendster. This indicates that a higher logging rate can easily be supported by using a NVMe SSD.

Table 9. Impact of Raw Text Data on Logging Phase

Graph Name	Logging Rate (in Million Edges/sec)	
	Binary Data	Raw Data
LANL	35.98	1.25
Twitter	82.62	3.10
Friendster	82.85	3.05
Subdomain	82.86	3.11
Kron-28	79.23	3.10
Kron-21	78.91	3.23

Logging rate is per-thread.

Recovery. Recovery only needs to perform the archiving phase at the bulk of the data. As we will show in Figure 16, the archiving is fastest when around 2^{27}–2^{31} edges are cleaned together. Hence, we take the minimum of this size as a recovery threshold to minimize the memory requirement of the IO buffer and the recovery time and get an opportunity to pipeline the IO read time of the data with recovery. Table 8 shows the total recovery time, including the 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 graphs 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 Data Ingestion Performance on Text Data

In this sub-section, we expect the incoming data in raw text format without having any vertex ID, such as LANL network logs. Ingesting raw text data involves parsing the information, and identified key information will be converted to vertex ID using a hashmap of GraphOne to create a binary edge. It will also need to parse the text to identify the edge weights that will also be stored within the binary edge. Here, the additional cost of parsing the data and interaction with hashmap will reduce the performance of the logging phase. So we only demonstrate the performance of logging phase, as performance of other phases will remain same.

We pick two kinds of text data to ingest. The *first* kind is in purely text format where the vertices of the edges are in the string format, and inserting them into GraphOne additionally requires interaction with hashmap, where the latter converts the strings to vertex ID. We pick LANL raw text data in this case. The hashmap is taken from the open-sourced Intel Thread Building Block library [70]. The*second* kind is just a pair of vertex IDs that only requires parsing the data, but no conversion to vertex ID is required. Therefore, interaction with hashmap is not required. We pick other graphs, including Twitter, Friendster, and so on, in a raw text graph to demonstrate the performance of logging phase. The second kind is selected to gain an intuition about the cost of vertex ID creation and lookup versus parsing cost.

Table 9 shows that parsing the raw text takes the most time for Kron-28 graph. For LANL graph, the parsing and interacting with the hashmap takes most of the time. The numbers in Table 9 are for the single-threaded logging phase. We can confirm that they do show sub-linear scaling with increasing thread count as the hashmap can support parallel ingestion and retrieval. Specifically, with eight threads for 8 LANL text files shows logging rate close to 5.55 million edges per second, a speedup of 4.44×. The sub-linear scaling is due usage of the atomic instructions while claiming the spot in the edge log.

However, we feel that it is not the ideal setup for parallel ingestions in the real world. For example, the actual data sources for GraphOne may not necessarily be the files; rather, they could be the database log changes, Kafka, and so on. In the case of Kafka, data are already sharded using a shared-nothing partitioning technique; therefore, we believe logging data from different shards to the same edge log to create a global order is not worth it, and separate edge logs would be the best way to go forward. We confirm that batching in independent edge logs shows linear scaling as per our testing.

7.3 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, Galois and static version of GraphOne are static graph systems, and LLAMA [59] as a snapshot system. Except for 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.

It should be noted that, unlike these systems, the main contribution of GraphOne is its data store. As the scope of this work is to show the advantage of our efficient data store, we try to mimic the implementation strategy of prior work as much as possible when comparing the analytics performance, thus motivating the need to integrate such data stores in the existing real-time analytics engines empirically. We also compare the data ingestion performance to demonstrate the superior design of GraphOne.

7.3.1 Snapshot-based System. This section compares graph batch analytics systems that create snapshots at bulk updates. We choose the in-memory LLAMA [59] system for the comparison, which is available in github as open source. Since LLAMA ingests data from files by using a *fread()* function call, which internally uses buffered IO system call to batch the edges. We create a *tmpfs* file-system so that the data read from files could be done faster to test data ingestion. We also modified GraphOne to do ingestion in the same way by calling *fread* for each edge from files hosted in tmpfs. We use the same set of graphs as listed in Table 7 with few differences because of the way LLAMA is implemented as discussed next. Though both of these differences are unfair to GraphOne, we could not find a better way of comparison.

First, LLAMA only takes a weighted graph with 4 bytes of weight but stores weights in a structure parallel to the adjacency store. So when BFS and PageRank runs, it does not access the weight structure at all. GraphOne stores the 4-byte weights in the adjacency store directly and hence will be accessed while running BFS and PageRank even though not required. The required input binary file is created from text files using a utility provided by LLAMA that inserts random 4-byte weights for each edges. Second, LLAMA also creates one snapshot for a batch of input that is presented as a file. Interestingly, LLAMA's multi versioned vertex array consumes a lot of memory, and therefore it cannot create even 512 snapshots for most of the graphs except a Kron-21 graph. So we choose to create 256 snapshots, but Kron-28 still runs out of memory, so we drop its comparison. In GraphOne, we create one snapshot for each 65,536 batch of edges (the archiving threshold), which will be equal to 32,768 snapshots for the Friendster graph.

Despite the advantage of LLAMA over GraphOne is this setup, GraphOne achieves huge speedup over LLAMA as shown in Figure 12 for data ingestion time, BFS, and PageRank. GraphOne achieves an average speedup of 6.38×, 1.66×, and 2.43× for data ingestion, BFS, and PageRank, respectively, for all graphs except Kron-21, for which the speedups are much higher at 31.47×, 24.79×, and 1.23×, respectively. There is just one exception of BFS performance in the Subdomain graph that has 140 levels in its BFS tree. LLAMA has a highly optimized BFS implementation as it uses, in addition, a bitwise status for each active vertex that avoids reading the actual BFS

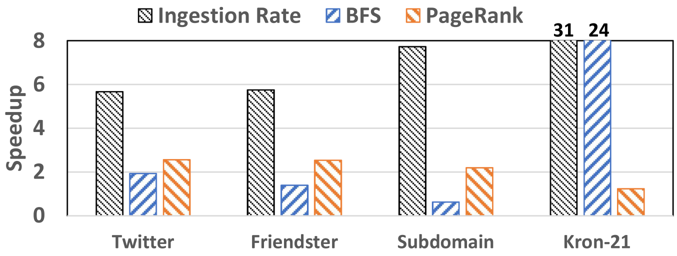


Fig. 12. Data ingestion rate and analytics performance comparison of GraphOne against LLAMA.

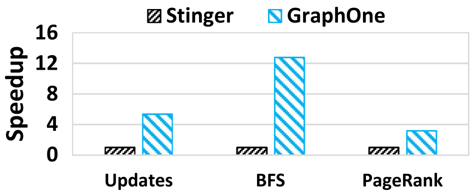


Fig. 13. Comparison against Stinger for in-memory setup.

algorithmic metadata for many vertices in so many iterations, and thereby GraphOne is slower by 33%.

The speedups of GraphOne are due to our memory management optimizations, overlap of logging and archiving, edge sharding technique, and avoiding the read of adjacency data through multi-versioned degree array. As we have demonstrated, memory management (Section 7.4.3) provides speedup to ingestion and analytics, while edge sharding (Section 7.4.4) leads to further improvement in data ingestion. The multi-versioned array in LLAMA is the main structure responsible for providing snapshot capability in LLAMA and is also used by analytics. However, GraphOne provides a snapshot through a multi-versioned degree array but uses a private degree array of a static view for the analytics that avoids reading the multi-versioned degree array completely. This technique apart from cacheline-/page-sized memory allocation further results in better analytics performance.

These advantages power GraphOne to perform better than LLAMA despite creating an order of more adjacency list snapshots and reading the unnecessary weights from the adjacency store for analytics. For results on the Kron-21 graph, we believe that overlapping of logging and archiving plays a major role, as each LLAMA snapshot is created using 2^{18} batched edges only (total edges/snapshot count) due to its smaller size. For other graphs, the batched edges per snapshot are much higher due to their larger sizes.

7.3.2 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 Reference [62] to compare the results on the RMAT graph.

Stinger is able to support 3.49 million updates per second on the same weighted RMAT graph, whereas GraphOne ingests 18.67 million edges per second, achieving 5.36× higher ingestion rate, as shown in Figure 13. 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 also implemented PageRank and BFS in a similar approach as Stinger. The comparison is plotted in Figure 13. Clearly, GraphOne is able to provide a better support for BFS and

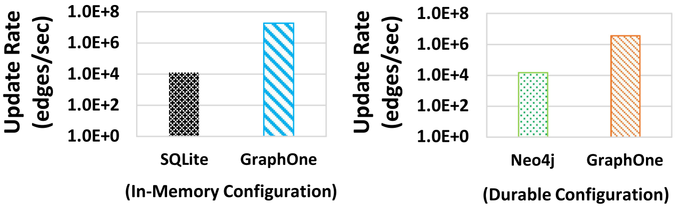


Fig. 14. Update rate comparison against SQLite and Neo4j.

PageRank achieving 12.76× and 3.18× speedups, 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.

7.3.3 Databases. GraphOne is only designed for a graphlike data structure and does not offer any other indexing apart from those that are created on source and destination vertices at this moment. Also, GraphOne offers snapshot isolation and not the strict serializability required for transaction support, hence GraphOne cannot compete against workloads that require transaction support. The value proposition here is that the adjacency store design can be integrated with the existing databases where prime focus is management and analytics on graph data but without transaction support, such as analytics queries. For example, the in-memory adjacency list in Neo4j [66] is optimized for read-only workloads, and new updates generally require invalidating and rebuilding those structures [72]. TitanDB [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. Adjacency store of GraphOne can be very useful in these products. Therefore, even if GraphOne is limited in this sense, we expect it to be beneficial by integrating it with graph databases and expect that their query engine can be integrated using our GraphView APIs to support their query languages.

Therefore, we compare against SQLite 3.7.15.2, a relational database, and Neo4j 3.2.3, a graph database for ingestion test only. SQLite and Neo4j support ACID transaction and do not provide native support for graph analytics. It is known that a 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 an experiment. The results are shown in Figure 14. 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 Reference [62]. 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 maintaining the graph data. Neo4j can ingest only 14.81K edges per second, whereas GraphOne ingests at 3.63M edges per second.

7.3.4 Stream Graph Engines. GraphOne runs stream computation and data ingestion to adjacency store (archiving phase) concurrently, while prior stream processing systems updates the adjacency store as part of stream computation where first operation is generally the update of the adjacency store as in Kickstarter [85], differential dataflow of Naiad [64], GraphBolt [61], GraphTau [34], and so on, except Kineograph [17] that we discussed in Section 3.2. This results into lower ingestion rate in prior work. In this sub-section, we will evaluate both the advantage of efficient data store, as well as the design choice of having the data store (adjacency store) updated in parallel to streaming analytics.

In this sub-section, we will first compare Kickstarter against the version of GraphOne that updates the adjacency store as a first operation of streaming analytics workflow followed by actual analytics computation as done in Kickstarter. This serves a baseline version of GraphOne for stream analytics comparison, and then we demonstrate the advantage of GraphOne over the baseline GraphOne by following our design decision of updating the adjacency store in parallel to analytics.

We use an open-source binary executable file of Kickstarter's single node multi-threaded implementation that has been shared by authors through github (source code is not available yet), and contains only BFS and SSSP (single source shortest path) implementation. It only prints the time taken by the analytics, but not the time taken by the insertion, so we estimate it using a wall clock (literally) by looking into some logs that are printed in the console. We choose BFS for the comparison, and it is naively implemented in GraphOne.

Kickstarter expects all the graph data in the text format, from which the base snapshot should be provided as a adjacency format, while the rest is in edge list format along with the number of batches to be created from this update data file. So we put these two files in a *tmpfs* file system to avoid the disk IO in the path of data read. Interestingly, Kickstarter, while performing streaming BFS very quickly, is very slow in updating the adjacency store. For the Kron-21 graph, we supplied half of the edges (16 million) as a base snapshot in the adjacency format, while the rest (16 million) are edge updates in the edge list format. The base snapshot creation and BFS are very fast, while the update insertion took slightly more than 210 minutes, and streaming BFS took 31 ms. The process runs for total 210 minutes and 26.948 s. We then repeated the same steps for the Twitter graph, but it did not finish running the process for over 18 hours. So we did not run any more experiments.

However, we supplied all the edges in edge list format in a text file hosted in the tmpfs file system and created the base snapshot at half the edge count and the next snapshot after all the edges arrived. The edges are inserted to GraphOne one edge at a time using the logging phase, while the archiving is controlled by the steaming analytics. The baseline GraphOne (i.e., archiving and BFS done in sequence) took 6.226 s to update the adjacency list (including the wait time until all the edges arrive), and 36 ms for BFS, and the total process time is 12.632 s. While our naive BFS is competitive to the Kickstarter's implementation, the adjacency store update for streaming edges is more than 2,000× faster than Kickstarter.

This experiment brings two observation: First, our data store is very optimized, and, second, such a data store can be easily integrated with Kickstarter to take advantage of the state-of-the-art graph stream processing and our highly optimized data store. We did not perform any additional experiment due to Kickstarter's poor update rate and the fact that their analytics can be implemented on top of GraphOne.

Advantage of Separating Archiving Phase from Stream Computation. We now show the advantage of running the adjacency store update (archiving phase) in parallel to stream computation. We report the wall clock time of the whole process in GraphOne compared to baseline GraphOne. Here baseline means that archiving is run as part of streaming workflow, as done in Kickstarter, and so on. In the setup, edges are always inserted one at a time using the logging phase. We take the above setup of providing the input data from a text file hosted in tmpfs containing data in the edge list format. This time BFS starts running as soon as the first snapshot of the adjacency store is available.

We first run GraphOne and figure out how many times streaming BFS was executed. This is because archiving runs in parallel to BFS, and when one streaming BFS converges in any snapshot (captured using stateful stream view), it immediately pulls the current snapshot by calling *update-sstream-view()* and performs streaming BFS again. Therefore streaming BFS may run fewer times than the archiving phase count. Hence, we print the count of how many times streaming BFS was executed. We then run same number of streaming BFS in the baseline GraphOne by creating

Table 10. Impact of Separating Archiving Phase out of Analytics Workflow

Graph Name	Baseline GraphOne	GraphOne
Twitter	987.45	764.58
Friendster	1236.78	1032.35
Subdomain	975.74	817.90
Kron-28	2687.60	1990.68
Kron-21	19.37	13.87

GraphOne, by this separation, decreases the overall runtime of data ingestion and analytics as shown in the third column of this table. The data are ingested from a text file hosted in a tmpfs file system. All the numbers are in seconds.

exactly that number of adjacency list snapshots. Table 10 shows the wall clock timing of the two systems.

The design decision to separate the archiving from the analytics workflow increases the overall average performance by 28.58% for graphs. It should be noted that GraphOne actually creates 29,954 adjacency store snapshots compared to 1,870 created by the baseline GraphOne, while both versions run streaming BFS 1,870 times. The improvement in time despite creating more adjacency list snapshot in GraphOne is due to continuous creation of adjacency store snapshots in parallel to stream analytics, which can schedule the stream analytics right at the time requested by pulling the latest snapshot using *update-sstream-view*. However, baseline GraphOne will incur a huge delay in scheduling the stream analytics, as it first has to create an adjacency store on the latest data.

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 the same for discrete stateful stream analytics. Thus, Table 8 is also an indication of their performance as well. We have also implemented streaming weakly connected components using ideas from COST [63] using stateless stream view APIs, and it can process 33.60 million stream edges per second on the Kron-28 graph.

Another advantage of separating the archiving phase out of streaming analytics is that now concurrent stream analytics can be executed from the same data store as we have discussed in Section 3.2 and Section 6.2. We explore this next.

Concurrent Execution of Stream Analytics. The separation of the data store as an abstraction provides benefit of executing many analytics together, such as multiple batch analytics or multiple stream analytics or any combination of both from the same data store, and therefore many complex usecases can be implemented using the GraphOne data store and GraphView APIs. As discussed in Section 3.2 and Section 6.2, prior graph analytics systems in theory can only execute multiple batch analytics but not multiple stream analytics from the same adjacency store. Even snapshot-based stream analytics systems such as Kinerograph [17] cannot execute together two classes of stream analytics—those that need to access whole data and those that requires window access—as they propose a different snapshot technique to be implemented in future (see Section 6.3 in the article [17]) for the latter access. Prior stream analytics can decide to deploy multiple instances of the same stream analytics engine to run concurrent stream analytics but will also produce a separate adjacency store. Therefore, we choose multiple stream analytics to demonstrate the impact of their execution from the same adjacency store.

We choose stream BFS starting from different source vertices, and each one creates a separate stateful stream view handle for execution. The choice of selection of streaming BFS is completely random, and we could have also picked up other stream analytics. We are only interested in

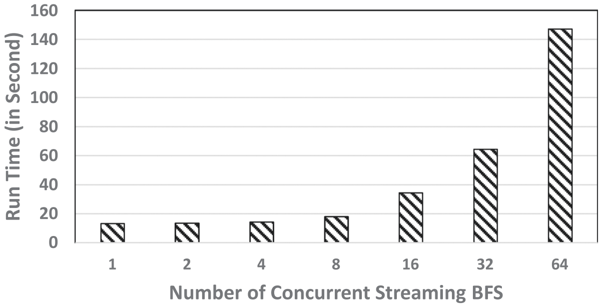


Fig. 15. Showing total time of GraphOne when different number of concurrent streaming BFS are executed while data are ingested in Kron-16 graph. All the data are ingested from a text file hosted in a tmpfs file system.

demonstrating this capability that is enabled by GraphOne, while other consequences, such as system overload due to multiple real-time analytics, are outside the scope of the article.

Figure 15 shows that it is possible to execute more concurrent streaming BFS while the data arrive in the Kron-21 graph. The actual streaming BFS computation takes much less time as they utilize the prior computation results, and the graph is getting inserted from a text file hosted in tmpfs, where parsing each text edge to convert it to a binary edge list format slows down the data logging rate (see Table 9). This implies that the system has sufficient CPU resources to run more streaming BFS. It is only when the count increases beyond 8 that each component (logging, archiving, and streaming BFS) starts to fight for CPU and memory bandwidth resources and that the overall system performance decreases.

It would also be possible to concurrently executed batch and streaming version analytics together as well, where the batch version will simply run on the latest data at the time whenever they are called by creating a static view and exiting after the computation is over, while streaming analytics will continue executing until unregistered.

Discussion on Staleness. It should be noted that the experiments in this section were run using *stale* stateful stream view. Although including the latest edges in the computation will incur some penalty (see Section 7.4) if a real-time nature is required, we claim that it is equally or more real time in nature than Kickstarter or baseline GraphOne. This is because by the time the adjacency store is ready in those systems to execute the stream analytics, the adjacency store is more or equally stale than the one used in GraphOne, as updating the adjacency store (archiving phase) takes some time (e.g., 6.22 s in Kron-21 for ingesting 16 million streaming edges to adjacency store), and then only actual analytics will start. However, GraphOne will supply the latest adjacency list that will be stale by at max non-archived edge count (which is always less than twice the archiving threshold) or will be no stale if slight lower analytics performance is acceptable.

7.4 System Design Parameters

In this sub-subsection, we present the various results associated with various system design parameters and individual techniques.

7.4.1 Performance Tradeoff in Hybrid Store. We first characterize the behavior of the hybrid store for different numbers of non-archived edges. Figure 16 shows the performance variation of archiving rate, BFS, PageRank, and 1-Hop query for the 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 a negligible drop in the

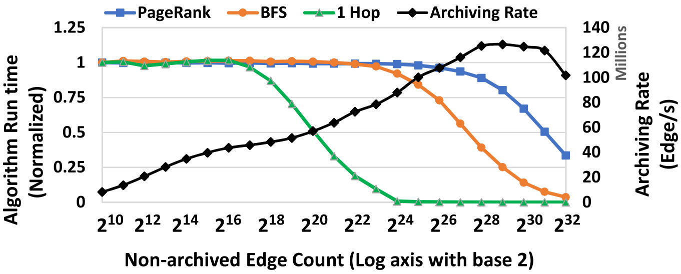


Fig. 16. Algorithmic performance and archiving rate variation for different non-archived edge count.

Table 11. Batch Analytics Performance (in Seconds) When There Are 0 or 1,024 Non-archived Edges in GraphOne for a Kron-28 Graph

Non-archived Edge Count	BFS	PageRank (5 Iterations)	1-Hop (512 Queries)
0	1.3101	32.1611	8.32×10^{-4}
1,024	1.3255	32.2803	5.47×10^{-2}

analytics performance. Hence, we recommend that the value of the archiving threshold should be 2^{16} edges, as the logging overlaps with the archiving. GraphOne is able to sustain an archiving rate of 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 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 remains same from 2^{10} up to 2^{17} and then drops as the number of non-archived edges becomes really large. This raises a question on whether there will be a performance drop in the analytics when we increase the edge count from 0 to 2^{10}. Table 11 shows that the drop is only applicable for 1-Hop query. This is because BFS and PageRank have already spawned multiple threads for execution from the adjacency store, and therefore they can easily process the edges in parallel in edge-centric processing (i.e., it only need to scan non-archived edges once each iteration), and thus the performance drop will be negligible. However, 1-Hop query is single threaded as getting neighbors of a vertex from an adjacency list does not require multiple threads. But either the cost to create threads for parallel scan or doing it sequentially will lead to performance drop. The Table 11 and Figure 16 shows the cost to process non-archived edges when we create multiple threads to take advantage of the parallelism offered by edge log scan (one time for each 1-Hop query). Scanning without parallelism will also reduce the performance for 1-Hop queries, but we have not measured it.

Figure 16 also shows that higher archiving threshold lead to a better archiving rate, e.g., a archiving threshold of 1,048,576 (2^{20}) edges can sustain an archiving rate of 56.99 million edges per second. The drawback is that the analytics performance will be reduced, as it will find more 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

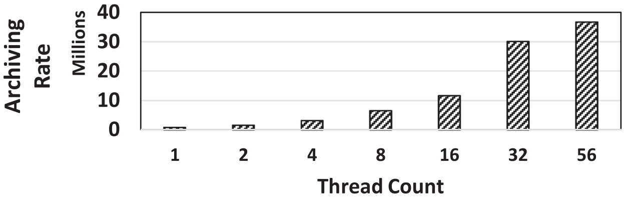


Fig. 17. Archiving rate scaling with thread count.

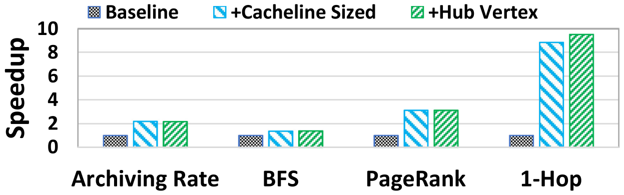


Fig. 18. Caheline-sized memory allocation brings huge performance gain, while hub vertex handling on top of cacheline-sized memory allocation improves the query performance only.

impact of large working set size that leads to more last-level-cache transactions and misses while filling the edge arrays.

7.4.2 Scalability. The edge sharding stage removes the need of atomic instruction or locks completely in the archiving phase, which results in better scaling of archiving rate with increasing number of threads as plotted in Figure 17. 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.

7.4.3 Memory Allocation. Figure 18 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 the Kron-28 graph, while speeding up BFS, PageRank, and 1-Hop query performance by 1.37×, 3.11×, and 8.82×, respectively.

Hub vertices handling additionally improves the query performance (by 7.5%). This was expected, as the number of hub-vertices in any power-law graph are very small, so performance improvement would be limited.

GraphOne without Any Memory Chain. Memory allocation techniques tries to minimize the memory link or chain counts, but it cannot reduces it to just one due to the evolving nature of input graph data. In this part, we show how much GraphOne suffers from such an ideal system if it can achieve such a goal. We design an ideal system, called static GraphOne, that has only one memory link.

Figure 19 shows this performance drop; specifically, by 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.

7.4.4 Advantage of Edge Sharding. Figure 20 shows the performance of the archiving phase with the edge sharding stage on the Kron-28 graph in comparison to an alternate approach where each

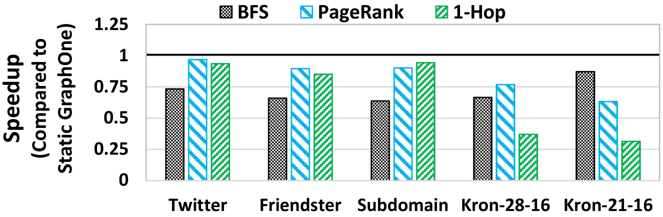


Fig. 19. Graph analytics performance in GraphOne compared to its static version, which has no chaining requirement.

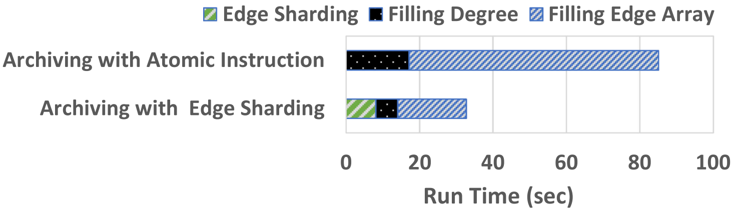


Fig. 20. Impact of the edge sharding on archiving phase performance.

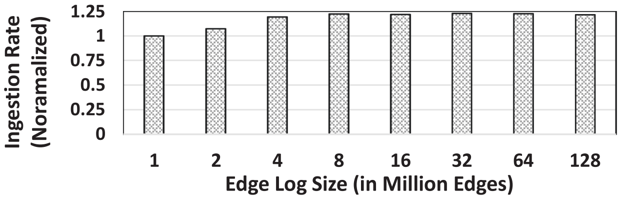


Fig. 21. Showing Ingestion rate when edge log size increases.

worker thread can directly work on the edge log by dividing the number of edges equally among them. This will require the use of atomic instructions when two threads compete to add the edges to the adjacency list of the same vertex. Also, the order of processing will not be the same as the order of arrival of updates, and thus the alternate method will suffer from the ordering issues that we have outlined in Section 5.

The edge sharding stage improves the whole archiving phase by 2.59× for the Kron-28 graph as shown in Figure 20. It should be noted that the degree array computation also shows speedup, as we removed the atomic instruction usage from this stage as well.

7.4.5 Edge Log Size. Figure 21 shows the effect of edge log size on overall ingestion rate on the Kron-28 graph. A smaller edge log would block the logging thread frequently, as the archiving rate is generally slower than the logging rate. Therefore, a larger edge log size can accommodate more edges, which will than force the archiving to run on large non-archived edges, which results in a higher archiving rate. Clearly, an edge log size greater than 4 million edges (32 MB) does not have any impact on overall ingestion rate.

7.4.6 Degree Array Snapshot Counts. GraphOne keeps many recent degree values of each vertex that are recycled after their number increases beyond a number. In this sub-section we show the performance implication by varying this number, before we propose its recommended value. Keeping more degree array in GraphOne provides a higher number of adjacency list snapshots, but managing them incurs more memory cost and results in a slower archiving rate as recycling of

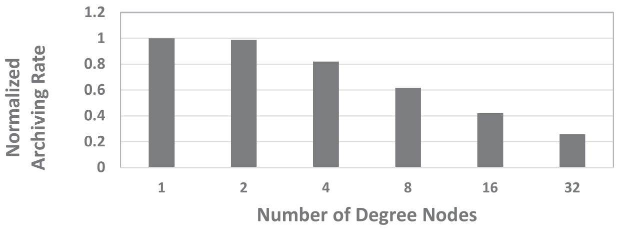


Fig. 22. Impact of the degree node count on archiving phase performance.

Table 12. BFS Performance When Using Low-level vs. High-level GraphView API

Graph Name	Low-Level API		High-Level API	
	BFS	PageRank	BFS	PageRank
Kron-28	15.09	58.62	15.51	60.17
Twitter	3.66	14.39	3.67	14.48

old degree nodes requires more pointer chasing in the degree array structure. Figure 22 shows this impact on the archiving rate for the Kron-28 graph. Clearly, having fewer degree array versions is better for data ingestions. Therefore, GraphOne keeps only three degree array versions and recommends using private views for long-running analytics.

7.5 GraphView

In this subsection, we measure the cost of creating different kinds of GraphViews and measure the performance impact of different parameters.

Cost of High-level API of GraphView. As discussed in Section 6, we provide two levels of API: the low-level API, *get-nebrs-archived-*()*, which directly allows access of references of the internal edge array blocks, and the high-level API, *get-nebr-**, which simplifies the job of programmer to implement the vertex-centric implementation and also makes the programmer not traverse all the links in the edge array. However, the high-level API has a cost, as it internally traverses and copies all the edge array blocks data for each vertex to a user-supplied buffer.

Table 12 shows the cost of BFS (push based) when using high-level APIs over low-level APIs. Clearly, simplifying the programmer's job does not have much cost. However, there could be a cost with high-level APIs in some cases, such as when implementing BFS using a direction-optimized technique [11] (i.e., using push+pull both). In this case, few levels of BFS are computed using the pull method where not every neighbors of active vertices are visited. Thus, high-level API will have obvious disadvantages, as it always copies the whole neighbor list to a user-supplied buffer. Clearly, this is a limitation of the GraphView API in its current form, as for direction-optimized BFS on a Kron-28 graph, high-level API has around 140% overhead. However, we do not see any limitation on extending the set of APIs, such as getting only the last k neighbors of a vertex, where k would be a user-supplied value. We leave this for future work.

Cost of Private Static View Creation. Passing *private* flag in the view creation API copies the degree array and non-archived edges to a private buffer. The copy takes some time but improves the working of GraphOne, as the copying makes the data management independent of analytics as former can recycle the degree array easily. Table 13 shows the cost of creating a private static view, which requires copying the degree array is in milliseconds, much less than BFS.

Table 13. Time Spent (in Milliseconds) in GraphView Creation

Graph Name	Kron-28	Kron-21	Twitter
View Creation Time	215.53	1.77	111.66

8 RELATED WORK

Static graph analytics systems [7, 16, 29, 30, 38, 45, 46, 53, 55, 57, 58, 67, 74, 75, 87, 89, 91, 94–96] support only batch analytics, where pre-processing consumes much more time than the computation itself [60], and are clearly not designed to support real-time analytics.

Grapchi [50] and other snapshot-based systems [28, 37, 41, 59, 71, 86] support bulk updates only. However, in Graphchi, the updates are not made visible to the graph until the currently running analytics finishes. This implies that updates can be ingested only in long intervals, and execution of another analytics is not supported while the first one is running, as the updates will hardly get a chance to be merged to the graph if one or the other analytics is always running. LLAMA [59] uses a multi-versioned vertex array and reverse chaining of a delta adjacency list to create multiple delta snapshots. However, it supports only bulk updates. Clearly, creating a new delta snapshot using an adjacency list at each fine-grained update will results in thousands of snapshots of the vertex array that must be kept for some time as deletion can only happen once computation finishes on the snapshot. Thus, soon the system will run out of memory and storage resources.

Chronos [28], e.g., also supports snapshots for bulk updates only and uses a bitmap for each edge in the edge array instead of a multi-version vertex array to identify whether an edge is part of a particular snapshot. The bit length assigned to each bitmap decides how many snapshots Chronos can manage, which clearly cannot support snapshots at fine-grained updates.

Naiad [65], a timely dataflow framework, supports iterative and incremental computation but does not offer the data window on the graph data. Other stream analytics systems [17, 34, 65, 81] support stream processing and snapshot creation, some offering a data window but mostly at bulk updates only. Stream databases [5, 6] provide only stream processing. TIDE [88] introduces probabilistic edge decay that samples data from a base store.

Graph data sharding [20, 45, 50, 51, 57, 92] is a well-studied problem. Our edge sharding is different, as our aim is to keep ordering intact in the adjacency store and remove usage of atomic instructions from archiving phase. Nonetheless, prior sharding techniques may be very useful for dividing the GraphOne data in many machines, which is left as a future work.

Prior works [26, 78] follow an integrated graph system model that manages online updates and queries in the database and replicates data in an offline analytics engines for long-running graph analytics tasks. As we have identified in Section 1, they suffer from excessive data duplication and the weakest component problem. Zhang et al. [93] also argue that such composite design is not optimal. GraPU [80] 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.

Many dynamic graph systems manage graph data at a read-committed isolation level only [21, 23, 79] using an atomic update construct without providing a streaming access or creation of snapshots. In other words, these systems modify the per-vertex edge arrays, while an analytics task is running. As a result, different iterations of the analytics may run on different versions of the dataset. Hence, the result can be semantically incorrect or the convergence may get delayed [85].

The in-memory adjacency list in Neo4j [66] is optimized for read-only workloads, and new updates generally require invalidating and rebuilding those structures [72]. Titan [2] (also known as Janus Graph), an open source graph analytics framework, is built on top of other storage engines, such as HBase and BerkeleyDB, and thus does not offer an adjacency list at the storage layer. OrientDB [68] builds a graph model using a document store model where all information of a vertex, including all of its neighbors, are stored in a document.

Key value stores are basic building blocks for many NoSQL data storage systems that support atomic updates only. However, the append operation or partial update of values are generally not possible in key-value store, so it is necessary to store graph data in an adjacency list format. Therefore, systems must develop a graph data layer on top of a key-value store, such as trinity [79], which defined an append operation using migration of an old value (neighbor list) of a source vertex (key), which may trigger a lot of migration. Therefore, Facebook also uses a specialized data store, called TAO [14], for storing its social graph by moving away from memcache key-value architecture. Hence, in the absence of a higher layer, a key-value store is not natively suited for a graph eco-system, while GraphOne provides native support of an adjacency list format with an in-built support for append operation.

Timestamp is a known method to provide snapshots in an adjacency list format [83]. As we have discussed, it is inefficient to maintain adjacency list snapshots at edge-level granularity and cannot provide streaming view. Also, additional storage for storing the timestamp will double the memory cost and will introduce additional complexity and performance degradation as each analytics and query need to be made timestamp aware.

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 our FAST'19 shepherd, Ashvin Goel, and the Usenix FAST'19 and ACM Transaction on Storage reviewers for their suggestions. Pradeep Kumar did part of this work as a graduate research assistant at George Washington University.

REFERENCES

[1] [n.d.]. Friendster Network Dataset—KONECT. Retrieved from http://konect.uni-koblenz.de/networks/friendster.

[2] [n.d.]. Titan Graph Database. Retrieved from https://github.com/thinkaurelius/titan.

[3] [n.d.]. Twitter (MPI) Network Dataset—KONECT. Retrieved from http://konect.uni-koblenz.de/networks/twitter_mpi. http://konect.uni-koblenz.de/networks/twitter_mpi.

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

[5] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, et al. 2005. The design of the borealis stream processing engine. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR'05)*, Vol. 5. 277–289.

[6] Daniel J. Abadi, Don Carney, Ugur Cetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. 2003. Aurora: A new model and architecture for data stream management. *VLDB J.* 12, 2 (2003), 12039.

[7] Zhiyuan Ai, Mingxing Zhang, Yongwei Wu, Xuehai Qian, Kang Chen, and Weimin Zheng. 2017. Squeezing out all the value of loaded data: An out-of-core graph processing system with reduced disk I/O. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC'17)*. 125–137.

[8] Leman Akoglu, Hanghang Tong, and Danai Koutra. 2015. Graph based anomaly detection and description: A survey. *Data Min. Knowl. Discov.* 29, 3 (1 May 2015), 626–688.

[9] Reka Albert, Hawoong Jeong, and Albert-Laszlo Barabási. 1999. Internet: Diameter of the world-wide web. *Nature* 401, 6749 (9 Sept. 1999), 130–131. DOI: https://doi.org/10.1038/43601

[10] Darko Anicic, Paul Fodor, Sebastian Rudolph, and Nenad Stojanovic. 2011. EP-SPARQL: A unified language for event processing and stream reasoning. In *Proceedings of the 20th International Conference on World Wide Web.* ACM, 635–644.

[11] S. Beamer, K. Asanovic, and D. Patterson. 2012. Direction-optimizing breadth-first search. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'12).*

[12] Bibek Bhattarai, Hang Liu, and H. Howie Huang. 2019. CECI: Compact embedding cluster index for scalable subgraph matching. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD'19).*

[13] Ulrik Brandes. 2001. A faster algorithm for betweenness centrality*. *J. Math. Sociol.* 25, 2 (2001), 163–177.

[14] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry C. Li, et al. 2013. TAO: Facebook's distributed data store for the social graph. In *Proceedings of the USENIX Annual Technical Conference.* 49–60.

[15] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A recursive model for graph mining. In *Proceedings of the SIAM International Conference on Data Mining (SDM'04).*

[16] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. 2015. PowerLyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the 10th European Conference on Computer Systems.* DOI: https://doi.org/10.1145/2741948.2741970

[17] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. 2012. Kineograph: Taking the pulse of a fast-changing and connected world. In *Proceedings of the 7th ACM European Conference on Computer Systems.*

[18] Sutanay Choudhury, Lawrence B. Holder, George Chin, Khushbu Agarwal, and John Feo. 2015. A selectivity based approach to continuous pattern detection in streaming graphs. In *Proceedings of the 18th International Conference on Extending Database Technology (EDBT'15).*

[19] James Cipar, Greg Ganger, Kimberly Keeton, Charles B. Morrey III, Craig A. N. Soules, and Alistair Veitch. 2012. LazyBase: Trading freshness for performance in a scalable database. In *Proceedings of the 7th ACM European Conference on Computer Systems.* ACM, 169–182.

[20] Dong Dai, Wei Zhang, and Yong Chen. 2017. IOGP: An incremental online graph partitioning algorithm for distributed graph databases. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing.* ACM, 219–230.

[21] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. 2019. Low-latency graph streaming using compressed purely-functional trees. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation.* ACM, 918–934.

[22] David Easley and Jon Kleinberg. 2010. *Networks, Crowds, and Markets: Reasoning about a Highly Connected World.* Cambridge University Press.

[23] David Ediger, Robert McColl, Jason Riedy, and David A. Bader. 2012. Stinger: High performance data structure for streaming graphs. In *Proceedings of the 2012 IEEE Conference on High Performance Extreme Computing (HPEC'12).* IEEE, 1–5.

[24] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. 1999. On power-law relationships of the internet topology. In *ACM SIGCOMM Computer Communication Review*, Vol. 29. ACM, 251–262.

[25] FlockDB. 2010. Retrieved from https://blog.twitter.com/engineering/en_us/a/2010/introducing-flockdb.html.

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

[27] Graph500. [n.d.]. Retrieved from http://www.graph500.org/.

[28] Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen. 2014. Chronos: A graph engine for temporal graph analysis. In *Proceedings of the European Conference on Computer Systems (EuroSys'14).*

[29] Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, Jeong-Hoon Lee, Min-Soo Kim, Jinha Kim, and Hwanjo Yu. 2013. 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.*

[30] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. 2012. Green-Marl: A DSL for easy and efficient graph analysis. In *ACM SIGARCH Computer Architecture News.*

[31] Yang Hu, Pradeep Kumar, Guy Swope, and H. Howie Huang. 2017. Trix: Triangle counting at extreme scale. In *Proceedings of the 2017 IEEE High Performance Extreme Computing Conference (HPEC'17).* IEEE, 1–7.

[32] Yang Hu, Hang Liu, and H. Howie Huang. 2018. TriCore: Parallel triangle counting on GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis.* IEEE Press, 14.

[33] Bernardo A. Huberman and Lada A. Adamic. 1999. Internet: Growth dynamics of the world-wide web. *Nature* 401, 6749 (1999), 131–132. DOI: https://doi.org/10.1038/43604

[34] Anand Padmanabha Iyer, Li Erran Li, Tathagata Das, and Ion Stoica. 2016. Time-evolving graph processing at scale. In *Proceedings of the 4th International Workshop on Graph Data Management Experiences and Systems.* ACM, 5.

[35] Hawoong Jeong, Bálint Tombor, Réka Albert, Zoltan N. Oltvai, and A.-L. Barabási. 2000. The large-scale organization of metabolic networks. *Nature* 407, 6804 (2000), 651–654.

[36] Yuede Ji, Hang Liu, and H. Howie Huang. 2018. iSpan: Parallel identification of strongly connected components with spanning trees. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. IEEE Press, 58.

[37] Xiaoen Ju, Dan Williams, Hani Jamjoom, and Kang G. Shin. 2016. Version traveler: Fast and memory-efficient version switching in graph processing systems. In *Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC'16)*. USENIX Association, 523–536.

[38] U. Kang, Hanghang Tong, Jimeng Sun, Ching-Yung Lin, and Christos Faloutsos. 2011. GBASE: A scalable and general graph management system. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.*

[39] Alexander D. Kent. 2015. Comprehensive, Multi-Source Cyber-Security Events. *Los Alamos National Laboratory.* DOI: https://doi.org/10.17021/1179829

[40] Alexander D. Kent, Lorie M. Liebrock, and Joshua C. Neil. 2015. Authentication graphs: Analyzing user behavior within an enterprise network. *Comput. Secur.* 48 (2015), 150–166. https://doi.org/10.1016/j.cose.2014.09.001

[41] Udayan Khurana and Amol Deshpande. 2013. Efficient snapshot retrieval over historical graph data. In *Proceedings of the 2013 IEEE 29th International Conference on Data Engineering (ICDE'13)*. IEEE, 997–1008.

[42] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2016. Ermia: Fast memory-optimized database system for heterogeneous workloads. In *Proceedings of the 2016 International Conference on Management of Data.* ACM, 1675–1687.

[43] David Knoke and Song Yang. 2008. *Social Network Analysis.* Vol. 154. Sage.

[44] Jay Kreps, Neha Narkhede, Jun Rao, et al. 2011. Kafka: A distributed messaging system for log processing. In *Proceedings of the International Workshop on Networking Meets Databases (NetDB'11).* 1–7.

[45] Pradeep Kumar and H. Howie Huang. 2016. 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'16).*

[46] Pradeep Kumar and H. Howie Huang. 2017. Falcon: Scaling IO performance in multi-SSD volumes. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC'17).* USENIX Association, 41–53.

[47] Pradeep Kumar and H. Howie Huang. 2017. SafeNVM: A non-volatile memory store with thread-level page protection. In *Proceedings of the IEEE International Congress on Big Data (BigData Congress'17).* IEEE, 65–72.

[48] Pradeep Kumar and H. Howie Huang. 2019. GraphOne: A data store for real-time analytics on evolving graphs. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST'19).*

[49] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is twitter, a social network or a news media? In *Proceedings of the Conference of the World Wide Web (WWW'10).*

[50] A. Kyrola, G. Blelloch, and C. Guestrin. 2012. Graphchi: Large-scale graph computation on just a PC. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'12).*

[51] Kartik Lakhotia, Rajgopal Kannan, Sourav Pati, and Viktor Prasanna. 2019. GPOP: A cache and memory-efficient framework for graph processing over partitions. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming (PPoPP'19).* ACM, New York, NY, 393–394. DOI: https://doi.org/10.1145/3293883.3299108

[52] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. 2005. Graphs over time: Densification laws, shrinking diameters and possible explanations. In *Proceedings of the 11th ACM SIGKDD International Conference on Knowledge Discovery in Data Mining.* ACM, 177–187.

[53] Heng Lin, Xiaowei Zhu, Bowen Yu, Xiongchao Tang, Wei Xue, Wenguang Chen, Lufei Zhang, Torsten Hoefler, Xiaosong Ma, Xin Liu, et al. 2018. ShenTu: Processing multi-trillion edge graphs on millions of cores in seconds. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis.* IEEE Press, 56.

[54] Hang Liu and H. Howie Huang. 2015. Enterprise: Breadth-first graph traversal on GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'15).*

[55] Hang Liu and H. Howie Huang. 2017. Graphene: Fine-grained IO management for graph computing. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17).*

[56] Hang Liu, H. Howie Huang, and Yang Hu. 2016. iBFS: Concurrent breadth-first search on GPUs. In *Proceedings of the SIGMOD International Conference on Management of Data.*

[57] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. 2012. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.* 5, 8 (2012), 716–727.

[58] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. 2017. Mosaic: Processing a trillion-edge graph on a single machine. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys'17).*

[59] Peter Macko, Virendra J. Marathe, Daniel W. Margo, and Margo I. Seltzer. 2015. LLAMA: Efficient graph analytics using large multiversioned arrays. In *Proceedings of the 2015 IEEE 31st International Conference on Data Engineering (ICDE'15).* IEEE, 363–374.

[60] Jasmina Malicevic, Baptiste Lepers, and Willy Zwaenepoel. 2017. Everything you always wanted to know about multicore graph processing but were afraid to ask. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC'17).* USENIX Association, Santa Clara, CA, 631–643. https://www.usenix.org/conference/atc17/technical-sessions/presentation/malicevic.

[61] Mugilan Mariappan and Keval Vora. 2019. GraphBolt: Dependency-driven synchronous processing of streaming graphs. In *Proceedings of the 14th European Conference on Computer Systems (EuroSys'19).* ACM, 25.

[62] Robert Campbell McColl, David Ediger, Jason Poovey, Dan Campbell, and David A. Bader. 2014. A performance evaluation of open source graph databases. In *Proceedings of the 1st Workshop on Parallel Programming for Analytics Applications (PPAA'14).* ACM, New York, NY, 11–18. DOI: https://doi.org/10.1145/2567634.2567638

[63] Frank McSherry, Michael Isard, and Derek G. Murray. 2015. Scalability! but at what COST?. In *Proceedings of the 15th Workshop on Hot Topics in Operating Systems (HotOS XV).*

[64] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. 2018. Differential dataflow. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR'18).*

[65] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: A timely dataflow system. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13).*

[66] Neo4j Inc. 2016. Retrieved from https://neo4j.com/.

[67] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A lightweight infrastructure for graph analytics. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'13).*

[68] Orient Database. 2016. Retrieved from https://orientdb.com/.

[69] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. The pagerank citation ranking: Bringing order to the web. Stanford InfoLab.

[70] James Reinders. 2007. *Intel Threading Building Blocks* (1 ed.). O'Reilly & Associates, Inc., Sebastopol, CA.

[71] Chenghui Ren, Eric Lo, Ben Kao, Xinjie Zhu, and Reynold Cheng. 2011. On querying historical evolving graph sequences. *Proc. VLDB Endow.* 4, 11 (2011), 726–737.

[72] Ian Robinson, Jim Webber, and Emil Eifrem. 2013. *Graph Databases.* O'Reilly Media.

[73] Daniel M. Romero, Brendan Meeder, and Jon Kleinberg. 2011. 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.* ACM, 695–704.

[74] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel. 2015. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'15).* ACM.

[75] A. Roy, I. Mihailovic, and W. Zwaenepoel. 2013. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'15).* ACM.

[76] Scott Sallinen, Keita Iwabuchi, Suraj Poudel, Maya Gokhale, Matei Ripeanu, and Roger Pearce. 2016. 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.* IEEE Press, 30.

[77] Jiwon Seo, Jongsoo Park, Jaeho Shin, and Monica S Lam. 2013. Distributed socialite: A datalog-based language for large-scale graph analysis. *Proc. VLDB Endow.* 6, 14 (2013), 1906–1917.

[78] Martin Sevenich, Sungpack Hong, Oskar van Rest, Zhe Wu, Jayanta Banerjee, and Hassan Chafi. 2016. Using domain-specific languages for analytic graph databases. *Proc. VLDB Endow.* 9, 13 (Sep. 2016), 1257–1268. DOI: https://doi.org/10.14778/3007263.3007265

[79] Bin Shao, Haixun Wang, and Yatao Li. 2013. Trinity: A distributed graph engine on a memory cloud. In *Proceedings of the SIGMOD International Conference on Management of Data.*

[80] Feng Sheng, Qiang Cao, Haoran Cai, Jie Yao, and Changsheng Xie. 2018. GraPU: Accelerate streaming graph analysis through preprocessing buffered updates. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC'18).*

[81] Xiaogang Shi, Bin Cui, Yingxia Shao, and Yunhai Tong. 2016. Tornado: A system for real-time iterative analysis over evolving data. In *Proceedings of the 2016 International Conference on Management of Data.* ACM, 417–430.

[82] Julian Shun and Guy E. Blelloch. 2013. 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'13).*

[83] Manuel Then, Timo Kersten, Stephan Günnemann, Alfons Kemper, and Thomas Neumann. 2017. Automatic algorithm transformation for efficient multi-snapshot analytics on temporal graphs. *Proc. VLDB Endow.* 10, 8 (2017), 877–888.

[84] M. J. M. Turcotte, A. D. Kent, and C. Hash. 2017. Unified host and network data set. In *Data Science for Cyber-Security.* World Scientific. DOI: 10.1142/9781786345646_001

[85] Keval Vora, Rajiv Gupta, and Guoqing Xu. 2017. Kickstarter: Fast and accurate computations on streaming graphs via trimmed approximations. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems.* ACM, 237–251.

[86] Guozhang Wang, Wenlei Xie, Alan J. Demers, and Johannes Gehrke. 2013. Asynchronous large-scale graph processing made easy. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR'13)*, Vol. 13. 3–6.

[87] Ming Wu, Fan Yang, Jilong Xue, Wencong Xiao, Youshan Miao, Lan Wei, Haoxiang Lin, Yafei Dai, and Lidong Zhou. 2015. GRAM: Scaling graph computation to the trillions. In *Proceedings of the 6th ACM Symposium on Cloud Computing.*

[88] Wenlei Xie, Yuanyuan Tian, Yannis Sismanis, Andrey Balmin, and Peter J. Haas. 2015. Dynamic interaction graphs with probabilistic edge decay. In *Proceedings of the 2015 IEEE 31st International Conference on Data Engineering (ICDE'15).* IEEE, 1143–1154.

[89] Da Yan, Yuzhen Huang, Miao Liu, Hongzhi Chen, James Cheng, Huanhuan Wu, and Chengcui Zhang. 2018. GraphD: Distributed vertex-centric graph processing beyond the memory limit. *IEEE Trans. Parallel Distrib. Syst.* 29, 1 (2018), 99–114.

[90] Kai Zeng, Jiacheng Yang, Haixun Wang, Bin Shao, and Zhongyuan Wang. 2013. A distributed graph engine for web scale RDF data. In *Proceedings of the 39th International Conference on Very Large Data Bases (PVLDB'13).* VLDB Endowment.

[91] Kaiyuan Zhang, Rong Chen, and Haibo Chen. 2015. NUMA-aware graph-structured analytics. *ACM SIGPLAN Not.* 50, 8 (2015), 183–193.

[92] Wei Zhang, Yong Chen, and Dong Dai. 2018. AKIN: A streaming graph partitioning algorithm for distributed graph storage systems. In *Proceedings of the 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing.* IEEE Press, 183–192.

[93] Yunhao Zhang, Rong Chen, and Haibo Chen. 2017. Sub-millisecond stateful stream querying over fast-evolving linked data. In *Proceedings of the 26th Symposium on Operating Systems Principles.* ACM, 614–630.

[94] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E. Priebe, and Alexander S. Szalay. 2015. FlashGraph: Processing billion-node graphs on an array of commodity SSDs. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15).*

[95] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A computation-centric distributed graph processing system. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'16).* 301–316.

[96] Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. 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.*

Received April 2019; revised August 2019; accepted September 2019