Dynamic Graph Storage: An Experimental Survey

Xiangyu Zhi[†], Xiao Yan[†], Keming Li[†], Bo Tang[†], Yanchao Zhu[‡], Minqi Zhou[‡]
[†]Department of Computer Science and Engineering, Southern University of Science and Technology
[‡]Gauss Department, Huawei Company
[†] {zhixy2021@mail., yanx@, likm2020@mail., tangb3@}sustech.edu.cn
[‡] {zhuyanchao2, zhouminqi}@huawei.com

ABSTRACT

Dynamic graph, which refers to the case that the graph constantly receives updates and needs to support real-time lookup and analytics, is common in many domains such as social networks and e-commerce. The underlying storage structure has a fundamental impact on dynamic graph applications and is studied by many works. However, existing works make different design considerations, yielding a variety of storage structures with complex trade-offs among different aspects, which makes it difficult to take lessons to design for new applications. To tackle this problem, we conduct an extensive survey and evaluation of existing graph storage systems, focusing on how the storage structures affect different aspects of performance, e.g., update throughput, lookup efficiency, algorithm execution time, and memory consumption. In particular, we discuss the storage structures of 14 representative systems in detail and characterize their design rationales and performance implications. We also summarize the lessons learned by listing the considerations/requirements one should clarify when designing a graph storage system and discussing how existing techniques can be composed to meet the requirements.

PVLDB Reference Format:
Xiangyu Zhi, Xiao Yan, Keming Li, Bo Tang, Yanchao Zhu, Minqi Zhou. Dynamic Graph Storage: An Experimental Survey. PVLDB, 16(1): XXX-XXX, 2023.
doi:XX.XX/XXX.XX

1 INTRODUCTION

Due to their excellent expressiveness, graphs are widely used as data representations in many domains including social networks [31, 33, 65], e-commerce [18, 23], and web analytics [10, 39, 53]. For a large number of applications, the graph keeps receiving updates at high velocity [25, 37, 59, 62], e.g., when users add each other as friends on social media or customers purchase products on an e-commerce website. In the meantime, lookups and graph analytic algorithms are expected to run efficiently on the latest version (or a snapshot) of the graph [48, 73], e.g., to check the friends of a user or to recommend products for the customers. These scenarios that require high-speed updates and real-time analytics are commonly referred to as the *dynamic graph* [19].

Proceedings of the VLDB Endowment, Vol. 16, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

The underlying storage structure plays a vital role in dynamic graph applications and is studied by a plethora of works [3, 22, 27, 28, 32, 43–45, 51, 54, 57, 58, 66, 71, 75, 76]. These works differ in their design considerations, including the data to store (e.g., only the graph topology or also complex vertex/edge properties), the level of data integrity (e.g., check update legality or not), and requirements of the specific application (e.g., allow to trace historical versions of the graph or not) [42, 64, 74]. As a result, they arrive at different storage structures, e.g., linked lists, variants of trees, arrays in different forms, and append-only logs. These structures in turn lead to complex trade-offs in different aspects of performance, i.e., update throughput, lookup efficiency, algorithm execution time, memory consumption, and thread scalability [56]. Thus, given a dynamic graph application, it is unclear what lessons we can learn from existing graph storage systems[1] to design a graph storage that suits the needs of the application.

To resolve the confusion, this paper conducts an extensive survey and experimental study of existing graph storage systems. We focus on *structural graph*, which includes the graph topology and some fixed length properties (e.g., weight) for each vertex/edge, but also cover some systems for *property graph* [12, 61], which may store complex and variable length properties for each vertex/edge. For 14 representative systems, we introduce how different structures are used to store the graph, and how key operations (e.g., edge insertion/deletion, reading the edges of a vertex and enumerating the entire graph) are performed on the structures. We also analyze the performance implications of different structures and the design rationales behind each system. To complement our analysis and better understand existing systems, we conduct extensive experiments to evaluate all aspects of their performance including update, read, memory and scalability. To summarize, we made the following contributions.

- We survey 21 graph storage systems and discuss 14 of them in detail. We also propose a taxonomy for existing graph storage systems, which classifies them into three categories according to the basic structures they use, i.e., *adjacency list variants*, *compressed sparse row (CSR) variants*, and *hybrid structures*.
- With extensive experiments, we provide comprehensive analyses on how the basic storage structures (e.g., linked list, trees and arrays) and key design choices (e.g., keeping sorted/unsorted edges and storing additional properties) affect performance.

[1]Some existing works should be called graph storage structures or schemes as they have not reached the complexity of a system, but the distinction can be subtle. Thus, we call all surveyed works graph storage systems for convenience.

- We summarize the lessons learned from existing graph storage systems and our experiments to guide future design. In particular, we list the key considerations/requirements one should clarify when designing a graph storage system and discuss how existing techniques can be used to meet these requirements.

Relation to existing surveys. Some surveys have been conducted on *distributed graph processing systems* (DGPSs) [52] and *graph databases* [17]. In particular, Lu et al. [50] surveyed DGPSs (e.g., Pregel [52] and Giraph [2]) by introducing their APIs and system designs for efficient distributed execution. Ammar et al. [11] conducted a more extensive survey for DGPSs, covering systems that adopt programming models different from the famous vertex-centric model. However, DGPSs typically work with static graph while we focus on dynamic graph. For graph databases, Lissandrini et al. [49] conducted a micro benchmark by testing their performance for basic CRUDT operations (i.e., create, read, update, delete, and traversal). Besta et al. [17] surveyed graph databases comprehensively by discussing their data models, storage structures, and query languages. Graph databases mainly handle property graphs while our survey focuses on structural graphs and their storage structures. The most related existing survey is [16], which discusses various aspects of dynamic graph systems, e.g., overall architecture, programming model and interface, storage, and support for incremental computation. Our survey is more focused on storage structure and comes with experiments to understand the performance implications of storage.

The rest of the paper is organized as follows. Section 2 introduces the workloads on dynamic graph and some basic graph storage structures as background. Section 3 discusses representative graph storage systems. Section 4 tests and analyzes the performance of existing graph storage systems with extensive experiments, and Section 5 summarizes the lessons learned from existing systems. Section 6 draws the concluding remarks and outlines future directions.

2 BACKGROUND

We mainly consider *structural graph* in the form of $G = (V, E)$, where V is the set of vertices and E is the set of edges. We assume that the graph is directed, and undirected graphs can be stored by treating each undirected edge as two directed edges. Each vertex $v \in V$ is indexed by a unique integer (called vertex ID), and each directed edge $e \in E$ from source vertex v to destination vertex u is indexed by a unique tuple $(v, u) \in V \times V$. Each edge $e \in E$ (resp. vertex $v \in V$) can have a weight w_e (resp. w_v), which is an arbitrary integer or float. Most works store the weight in-line with each vertex/edge, and thus we do not discuss how a system stores weight unless it adopts other methods. We also discuss some works that store *property graph* (PG)[12], where each vertex (resp. edge) has one additional label and an arbitrary number of properties (e.g., age and gender for a vertex with label person), because some applications may need to store complex properties along with the graph topology. We refer to the outgoing neighbors (resp. edges) of a vertex v as v's neighbors (resp. edges). Most works store the graph in the main memory and on a single machine, and we will be explicit if a work considers different scenarios (e.g., disk, GPU or distributed storage).

2.1 Workloads and Performance Metrics

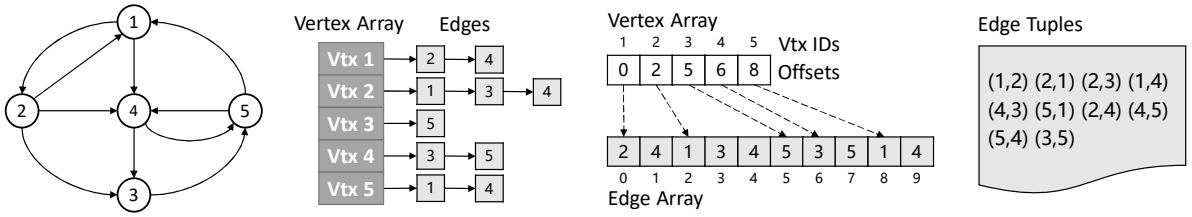
Workloads on dynamic graph can be classified into two main categories, i.e., *update* and *read*. Updates include *vertex update* (i.e., insert or delete vertices), *edge update* (i.e., insert or delete edges), and *property update* (i.e., change the weight or properties of a vertex or edge). Most works focus on edge update, which changes the graph G to a new graph G' by applying a set of updates $\mathcal{U} = \{e_1^+, e_2^-, \cdots, e_K^+\}$ (where e^+ indicates an edge to insert and e^- indicates an edge to delete). Property update is usually easier than edge update as it only requires to locate the properties (which is similar to edge lookup) while edge update also needs to allocate/de-allocate memory. Vertex insertion is also easy by appending to the storage structure for vertices and inserting the corresponding edges. Vertex deletion is more complex as it requires removing not only the outgoing edges of the target vertex v but also the incoming edges that point to v (which may span many source vertices). Most existing works do not discuss vertex deletion, possibly because it is rare in real applications [13, 21]. If efficient vertex deletion is required, existing works can be extended by recording the incoming edges for each vertex to facilitate fast removal. Update can be performed either in a *streaming* manner [55, 63, 70] (i.e., one update at a time) or in *batches* [7, 8, 28] (by accumulating some updates and conducting them together). The performance metric for update is *throughput*, which measures the average number of updates that can be conducted per second.

Algorithm 1: A general framework for graph algorithms

```
Input: The graph G = (V, E)
Output: The result Res(V)
/* Res(V) is vector that contians a number or
   data structure for each vertex v ∈ V        */
1 Res(V) = Initialize(G);
2 while Terminate(G, Res(V))==False do
3     for v ∈ V that passes filerting condition F_v(v) do
4         for edge e = (v, u) ∈ E that passes condition F_e(e) do
5             Res(v), Res(u) = Update (Res(u), Res(v));
6         end
7     end
8 end
```

Reads can be classified into *point* and *graph* queries according to their scopes. Point queries access information related to a single vertex/edge, e.g., verifying whether a vertex/edge exists, fetching the weight/label/properties of a vertex/edge, and scanning all neighbors of a vertex v, and etc. To make point queries efficient, the storage should support locating a vertex/edge with low cost and store the edges of a vertex together. Graph queries mainly refer to graph analytics algorithms that access a large part of the graph, e.g., PageRank (PR) [26, 60, 68, 72], label propagation (LP), breadth-first search (BFS), connected components (CC), triangle counting (TC), and single source shortest path (SSSP). We provide pseudo codes for these algorithms in our technical report [6] and summarize them as the general framework in Algorithm 1.

As shown by Algorithm 1, graph analytics algorithms usually run iteratively until a termination condition is satisfied. In each



(a) Graph **(b)** Adjacency List **(c)** Compressed Sparse Row **(d)** Edge List

Figure 1: An example of three basic graph storage formats

iteration, a vertex filtering condition $F_v(v)$ is used to determine the vertices to access, and for each qualified vertex, an edge filtering condition $F_e(e)$ is used to determine the edges to access. According to $F_v(v)$ and $F_e(e)$, we classify graph analytics algorithms into three categories, which have different requirements for graph storage.

- *Global traversal algorithms*, for which both $F_v(v)$ and $F_e(e)$ are null. In other words, these algorithms read all edges of the graph in an arbitrary order, and examples include PR and LP. They are efficient if all edges are stored contiguously in memory such that cache miss is minimized.
- *Local traversal algorithms*, for which $F_e(e)$ is null but $F_v(v)$ is not. That is, these algorithms access all edges of some specific vertices in each iteration, and examples include BFS, SSSP, and CC. They are efficient if the edges of the same vertex are stored contiguously in memory.
- *Fine-grained traversal algorithms*, for which both $F_v(v)$ and $F_e(e)$ are not null. These algorithms access some edges for some specific vertices in each iteration. One such example is TC, which, for a vertex v, only reads outgoing edges that point to destination vertices with larger ID than v. TC is efficient if the edges of the same vertex are stored contiguously and sorted according to their destination vertex IDs. Sorted edges also make edge lookup efficient and update legality check (i.e., if an edge to delete exists) efficient.

For read operations, the main performance metric is *running time*, especially for graph analytics algorithms. *Space consumption* is also important, which is the memory taken to store a graph and usually measured by the blowup compared to *compressed sparse row (CSR)*, a compact storage format that will be introduced shortly. Many works also consider *scalability*, where vertical scalability measures how update throughput and algorithm running time scale with the number of threads, and horizontal scalability measures how the two performance metrics scale with the size of the graph. Besides accessing the latest graph, some applications may require to access the graph up to some point in time or guarantee that a read operation should see no updates submitted after it. In these cases, timestamps are stored with each vertex/edge to indicate when they are updated.

2.2 Basic Graph Storage Formats

We introduce three basic graph storage formats, i.e., *adjacency list*, *CSR*, and *edge list*, which serve as the foundation for further discussion as most works either enhance (one of) them or combine their features. Figure 1 shows how a graph is stored using the three formats as example.

Adjacency list stores the edges of each vertex v as a linked list and uses an array to store the head of the linked lists for all vertices. In a linked list, each element contains the ID of the destination vertex and a pointer to the next element. The advantages of adjacency list are that it is easy to parallelize update/read for different vertices, and memory can easily be allocated/deallocated when inserting/deleting edges. The disadvantages include large space consumption due to the pointers and high cost to access all edges of a vertex due to the case misses caused by pointer chasing.

CSR uses a vertex array and an edge array to store a graph. The i-th element in the vertex array records the starting position of the edges of vertex v_i in the edge array, and the edges of each vertex are stored contiguously and sorted in the edge array. CSR is widely utilized to store static graphs due to its small memory footprint and efficient read (as the edges are contiguous in memory). However, CSR is extremely inefficient for updates as the entire vertex/edge array needs to be adjusted for a single edge insertion/deletion.

Edge list stores a graph as a set of unordered edge tuples. It is very efficient for edge update, which is conducted by appending a record (with a deletion indicator for edge deletion). However, reading the neighbors of a vertex (or a specific edge) is expensive as it requires to scan the entire edge list.

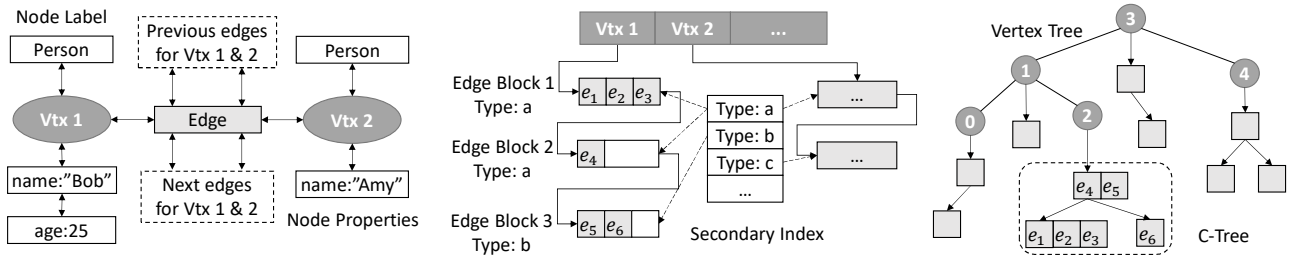
Adjacency matrix, which stores an n-vertex graph using an $n \times n$ matrix and uses 1 to indicate edges in the matrix, is also a well-known graph storage format. It is seldomly used in related works as they usually consider large graphs that are extremely sparse, for which adjacency matrix has very high memory consumption.

3 DYNAMIC GRAPH STORAGE SYSTEMS

Table 1 summarizes 21 existing dynamic graph storage systems and their key features. According to their basic storage structures, we classify them into three categories, i.e., *adjacency list variants*, *CSR variants* and *hybrids*. Adjacency list variants store the edges of different vertices in separate structures; CSR variants store the edges of many vertices in a shared structure; hybrids jointly utilize different basic storage structures to enjoy their benefits. In the following, we introduce and discuss representatives of each category.

3.1 Adjacency List Variants

These systems usually store the edges of different vertices separately using pointer-based structures (e.g., linked list, linked list of blocks,



(a) Neo4j storage format **(b)** Stinger storage format **(c)** Aspen storage format

Figure 2: An illustration of representative graph storage systems in the *adjacency list variants* category

Table 1: A summary of 21 dynamic graph storage systems, *PG* means property graph and *SG* means structural graph, *AL* stands for adjacency list and *EL* stands for edge list, *Sorted* refers to whether the edges of each vertex are sorted and *Index* means the indexes built for vertex/edge lookup

System	Model	Basic Struct	Sorted	Index
Neo4j[3]	PG	AL	No	None
Stinger[32]	SG	AL	No	Hash
Aspen[28]	SG	Tree	Yes	C-Tree
RisGraph[35]	SG	CSR	No	Hash
Sparksee[54]	PG	Tree	Yes	B+ Tree
Tegra[43]	PG	Tree	Yes	ART
LLAMA[51]	SG	CSR	No	None
Grace[58]	SG	CSR	Yes	Hash
LiveGraph[75]	PG	CSR	No	None
PCSR [71]	SG	CSR	Yes	None
GPMA [66]	SG	CSR	Yes	None
Teseo [27]	SG	CSR	Yes	ART+Hash
CSR++ [36]	PG	CSR	Yes	None
GraphOne [45]	SG	AL+EL	No	None
ZipG [44]	PG	AL+EL	Yes	Hash
A1 [22]	PG	CSR+Tree	No	B-Tree
Terrace [57]	SG	CSR+Tree	Yes	B-Tree
GridGraph [76]	SG	EL	No	None
GraPU [67]	SG	AL	No	Hash
Weaver [30]	PG	KV	No	Hash
Graphflow [38]	PG	CSR	Yes	None

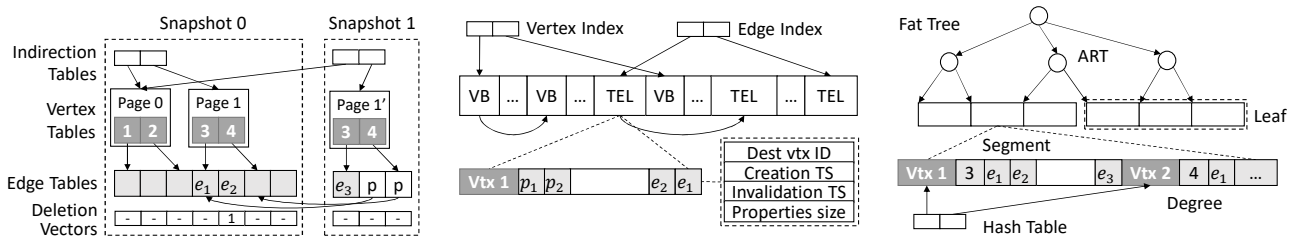
or variants of tree), and each element (e.g., leaf in the tree) in the structures are typically aligned with cache line size for efficient memory access. In particular, if the edges of a vertex are sorted, trees are usually adopted to maintain the order during updates; otherwise, linked list or linked list of blocks are used for the efficient update. These systems inherent the trade-offs of linked list: the advantages include easy parallelization for different vertices, and simple update procedure, and flexible schema (which is important for storing variable-length properties); the drawback is expensive graph traversal operations due to pointer chasing, albeit that using large block alleviates this problem.

Neo4j is a transactional graph database designed to store data on the disk. As shown in Figure 2(a), Neo4j extensively uses pointers in its storage structure. In particular, each edge stores two pointers to its source vertex and destination vertex, four pointers to the previous and next edges of the source vertex and destination vertex. Each vertex stores a pointer to the first edge in its adjacency list. The properties are stored in a doubly linked list of blocks with a block size of 41 bytes, and each vertex/edge stores the pointer to the head of its property linked list. All data (i.e., vertex, edge, and property block) in Neo4j have fixed size and different kinds of data are stored in separate files, which enables to calculate the physical location of an entry using its ID.

Property and edge insertions are conducted by adding new entries at the head of their respective linked lists. Read and deletion need to scan the linked list to look up the target edge/property. For analytical algorithms that enumerate all edges of a vertex, Neo4j has a high cost due to pointer chasing. However, the pointer-rich structure makes it easy to store vertices/edges having different schemas and thus a number of properties, and support ad-hoc graph traversal queries (e.g., accessing vertices with an edge that passes certain property filtering), which are common for graph databases.

Stinger uses linked list of blocks as shown in Figure 2 (b). The vertices are organized in an array (called logical vertex array, LVA) and can be accessed by their IDs. Each vertex holds a pointer to the linked list of edge blocks, and each block in the linked list contains a fixed number of slots (32 by default and configurable by the user) for edges with the same type (i.e., label). Edges with different types are stored in separate blocks even if some blocks are not full. e.g., block 2 in Figure 2 (b). The properties are stored in line with each edge including destination vertex ID, edge type, edge weight, and two timestamps. The timestamps are used for accesses-based time semantics, for example, reading the graph up to a point in time or fetching updates to the graph within a time interval. Each edge block also stores metadata including the lowest and highest timestamps of the edges and the high-water mark of the valid edges within the block. To access all edges with a given type, Stinger maintains an index called edge type array (ETA), which uses edge type as the key and maps to all edge blocks with the type.

To insert an edge, Stinger first scans the linked list to check whether the edge already exists. If so, the existing edge is updated with new properties; otherwise, the new edge is appended to an edge block of its type. If existing blocks do not have space, Stinger allocates a new block and adds it to the ETA. Edge deletion is conducted by first scanning the linked list to find the target edge and then removing it, and the freed space is reused for new edges. In summary, Stinger uses large blocks in the linked lists to reduce



(a) LLAMA storage format **(b)** LiveGraph storage format **(c)** Teseo storage format

Figure 3: An illustration of representative graph storage systems in the *CSR Variants* category

pointer chasing and improve traversal performance but update is slow as all edges of a vertex are scanned.

Aspen uses tree to store both the vertices and the edges of each vertex as shown in Figure 2(c). Specifically, the vertices are stored in a binary tree according to their IDs, and each tree node corresponds to a vertex. Each vertex stores the pointer to the root of a *compressed tree* (i.e., C-tree), in which each tree node is a block that stores some of its edges. C-tree is a search tree as it keeps the edges sorted, and in each tree node, the edges are compressed with delta encoding and encoded via byte code [69]. Each node in a C-tree is constrained to be larger than 32 bytes but smaller than a threshold with high probability. Re-balancing is conducted when the size constraints are violated. Note that Aspen stores only graph topology and does not support additional properties including edge weight.

The edge update operations (both insertion and deletion) are conducted in batches by merging C-Trees. For each batch of updates, Aspen first sorts the edges according to their destination and source vertices, and then builds a C-Tree for update edges with the same source vertex. After that, updates are performed by merging two C-Trees (one for existing edges and one for update edges) for each vertex. This batch optimization enables Aspen to achieve high update throughput at large batch size. C-tree compresses the edges and keeps them sorted, and thus Aspen has small memory footprint and good read performance. However, Aspen cannot support applications that require vertex/edge properties.

RisGraph stores the vertices using an array and the edges of each vertex using a dynamic array. Edge insertion is conducted by adding the new edge at the end of the dynamic array of the source vertex. When deleting an edge, RisGraph marks it as a tomb edge, and recycles the space when resizing the dynamic array. RisGraph conducts resizing when a dynamic array is full or the *fill factor* (i.e., the number of live edges divided by total number of slots) is low, and doubles or halves the array each time. As the edges of a vertex are not sorted, a full scan is required to look up a specific edge, which is expensive for high degree vertices. To tackle this problem, RisGraph builds a hash index for high degree vertices (with more than 512 edges by default), which is used to look up the offset of an edge in the dynamic array using destination vertex ID.

Beside the aforementioned systems, some other systems also adopt adjacency list structure. Like Neo4j, Sparksee [54] is a graph database and designed for disk storage. Sparksee models vertices, edges and properties as key-value pairs in the form of <*object*, *identifier*> and indexes them by separate B-Trees. Each vertex uses a bitmap to record its edges, and the bit map is compressed to reduce space consumption. Tegra [43] uses a persistent version of the adaptive radix tree (pART) [47], which adds persistence to ART by conducting path copying. In particular, Tegra uses one pART to store the vertices (with vertex ID as key) and one pART to store the edges (with source and destination vertices as key). The leaves of the pARTs store pointers to the vertex/edge properties. The edges of a vertex are fetched via prefix matching in the edge pART. The benefits of pART include efficient update and range scan.

3.2 CSR Variants

CSR variants usually store the edges of different vertices together like CSR but reserves empty space for efficient update. This makes graph traversal efficient by reducing cache miss but locking may be required when updating the edges of different vertices. If the edges are unsorted, dynamic array is usually used as the underlying storage structure; otherwise, data structures such as PMA are used to maintain the sorted order.

LLAMA stores a graph as a series of snapshots, and each snapshot adopts a structure similar to CSR as shown in Figure 3 (a). Specifically, each snapshot stores updates to the graph within a period of time and consists of three parts, i.e., *indirection array*, *vertex table* and *edge table*. Each element in the indirection array corresponds to a block of vertices; if a block of vertices receives no updates in the current snapshot, its indirection element stores the pointer to the same vertex block in the latest snapshot that has updates. This design reduces space consumption by skipping vertex blocks without changes. If a vertex block has updates, its vertices and inserted edges are stored using a format like CSR. The difference from vanilla CSR is that after the last edge of each vertex, LLAMA stores a pointer to the first edge of the vertex in the previous snapshot. This allows to scan all edges of a vertex by following the pointers across the snapshots.

For each batch of updates, LLAMA builds a snapshot for the edges to insert. For the edges to delete, LLAMA marks them as deleted via a bit map in the snapshots that contains them. LLAMA allows to merge snapshots by removing deleted edges and gathering the edges of a vertex in different snapshots. Update, especially deletion is not efficient for LLAMA as it requires to tracing the snapshots to find the edge. Read and graph traversal also suffer from cache misses when accessing the edges of a vertex via pointers. However, the

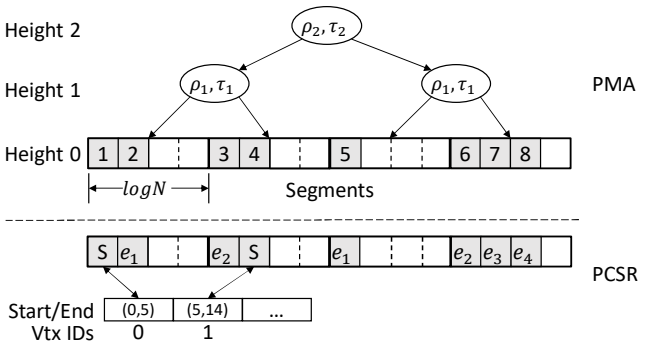


Figure 4: The structure of PMA and PCSR

snapshots allow LLAMA to keep the graph up to some predefined points in time.

Grace also uses snapshots of CSR like LLAMA. In each snapshot, Grace is similar to CSR and stores the vertices in a vertex log and the edges in an edge log. As the vertex IDs in the vertex log are not necessarily continuous, an edge pointer array is used in each snapshot to map each vertex to the offset of its edges in the edge log. Grace creates a snapshot for each batch of updates, and copies the edge pointer arrays of all previous snapshots to a new snapshot such that a vertex can access its edges in previous snapshots. Note that the edge logs are not copied to reduce space consumption. To parallelize execution over a multi-core CPU, Grace partitions the graph by source vertex ID and assigns each core to handle a segment of source vertices. For an inserted vertex, Grace uses a tailored algorithm to determine which core it should go to.

LiveGraph stores each vertex in a vertex block (VB) and the edges of each vertex in a transaction edge logs (TEL) as shown in Figure 3 (b). The VBs and TELs are organized together in an array. Both VB and TEL adopt a copy-on-write structure, and links to their previous version. The latest version of each VB and TEL can be found the ID of the (source) vertex via hash indexes, i.e., the vertex index and edge index in Figure 3 (b). Each edge keeps a destination vertex ID, two timestamps (one for creation and one for invalidation) and a property size at the tail of its TEL, and the actual properties at the end of the TEL. TEL reserves space in the middle for updates and scales its size by twice once it is full.

Before conducting edge updates, LiveGraph uses a bloom filter to check if the edge to delete/insert already exists. To insert an edge, LiveGraph appends its information (e.g., destination vertex ID) to the head of the TEL of the source vertex and properties to the end of the TEL. If the edge already exists, the invalidation timestamp of the old edge and the creation timestamp of the new edge are both set as the current time. To delete an edge, LiveGraph sets its invalidation timestamp to the current time to indicate deletion. LiveGraph can truncate the version linked list and recycle deleted edges for compaction. Overall, the read of LiveGraph performance suffers from the additional information it stores (e.g., timestamps). However, the timestamps allow LiveGraph to support time semantics and an explicit property size enables each edge to have a variable number of properties.

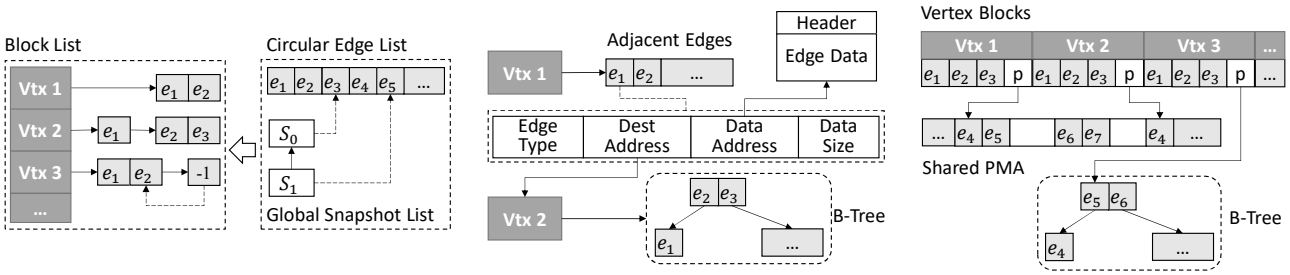
PCSR is based on the packed memory array (PMA) [14, 41], which we briefly introduce as its variants are widely used for graph storage. The upper part of Figure 4 shows an example of PMA. Specifically, PMA is an array that keeps the elements sorted and reserves empty slots for efficient insertion/deletion. For an array with N elements, PMA divides it into segments with length $O(\log N)$ and maintains an implicit binary tree by treating the segments as leaves. In the binary tree, each non-leaf node covers all leaf segments in its subtree, and comes with an upper bound (ρ) and lower bound (τ) for the overall fill factor of its segments. To conduct updates, PMA uses a binary search to find the position for insertion/deletion. If the fill factor bounds are violated after an update, a re-balancing operation is triggered, which first recursively finds a tree node, within which the fill factor bounds hold, by traversing the tree bottom-up, and then reallocates the elements among the segments covered by the tree node to satisfy all fill factor bounds. The amortized update cost of PMA is $O(\log N)$ on average and $O(\log^2 N)$ in the worst case [15].

As shown in the lower part of Figure 4, PCSR essentially adds a vertex array and uses PMA as the edge array in CSR. The edges of each vertex are sorted in PMA, and each element in the vertex array marks the starting and ending positions of the edges of the vertex in PMA. The 'S' mark at the starting position for each vertex is a pointer that links back to the vertex array, which is used to modify the vertex array when adjusting the PMA. Although PCSR's open-source implementation does not support edge deletion, it is due to engineering effort rather than the structural limitation of PMA. PCSR achieves efficient update by using PMA, and traversal is also quick by keeping all edges in an array. The drawbacks include larger memory consumption as PMA is not entirely filled and the requirement for locking when accessing each segment.

GPMA adapts PMA for dynamic graph storage on GPU, which significantly outperforms CPU in computation power and memory bandwidth. As GPU has far more threads than CPU, GPMA focuses on parallelizing PMA update operations and proposes two update schemes. The first is a workload-parallel scheme, where each thread takes one edge update, checks for the segments that need to be locked for the update (may lock multiple segments due to re-balancing), tries to acquire the locks until successful, and finally conducts the update. As this scheme hinders parallelism due to lock contention among the threads, a lock-free scheme called GPMA+ is proposed, which first organizes updates that lock the same segments into a group and then conduct updates for independent groups that lock different segments concurrently.

Teseo organizes several segments in PMA as a leaf in the fat tree (a variant of B+ Tree) as shown in Figure 3 (c). A leaf is a basic unit for locking and resizing, and when the size of a leaf exceeds a given threshold C_L, it will be split into two leaves. In each segment, Teseo stores the vertices in increasing order of their IDs, and each vertex is followed by its number of edges and the destination vertices of the edges. Each segment also records the smallest and largest vertex IDs in it as metadata. Teseo uses an ART as the primary index to locate the starting segment of each vertex (the data of a vertex may span multiple segments), and a hash table as the secondary index for each segment to locate the position of a vertex in the segment.

Update operations of Teseo resemble PCSR but have some differences. First, when a segment is full, Teseo adopts an edge list to host the inserted edges to delay the resizing operation. Second, Teseo



(a) GraphOne storage format **(b)** A1 storage format **(c)** Terrace storage format

Figure 5: An illustration of representative graph storage systems in the *hybrid structure* category

does not actively re-balance a leaf when its fill factor is low due to edge deletions. Instead, a service thread is used to periodically visit the fat tree to merge small leaves. In summary, Teseo improves PCSR by reducing the range and frequency of re-balance operations for lower update costs.

Some other systems also adopt CSR-style structures. CSR++ [36] adopts a block-based CSR structure, which partitions the vertex array of CSR into blocks and uses a dynamic array to store the edges of each vertex block. This design narrows down the range of adjustment during updates from the entire structure to a block.

3.3 Hybrid Storage Systems

Hybrid systems jointly use multiple basic structures to store a graph. As different basic structures excel in different aspects, hybrid systems can enjoy the advantages of multiple structures. The key design choices are how to combine the basic structures and transform the graph across different representations.

GraphOne jointly utilizes the adjacency list and edge list as shown in Figure 5(a). The adjacency list is used to store archived edges and is a linked list with block size aligned with the cache line. The edge list is used to store edge updates and implemented as a circular buffer. Both inserted and deleted edges are appended to the edge list, and a special mark is used for deleted edges. Updates in the edge list are merged into the adjacency list when instructed by the user or the edge list reaches a certain size. To conduct the merge, the edge updates are first grouped according to their source vertex ID and then added as new blocks to the rear of the adjacency lists. GraphOne tracks different snapshots of the graph, and thus edge updates in different snapshots are stored in separate blocks even if some blocks are not full. By combining the adjacency list with the edge list up to some point in time (e.g., S_0 and S_1 in Figure 5(a)), GraphOne allows to access the graph at specific time points using its GraphView interface. In summary, GraphOne allows quick update and fine-grained version tracking using an edge list, and efficient access to the edges of a vertex using an adjacency list. However, graph traversal can be inefficient if the edge list is large.

ZipG builds on Succinct [9], a distributed data store that supports random access and sub-string search on compressed unstructured data and key-value pairs. The archived vertices and edges are stored in two flat unstructured files, i.e., Nodefile and EdgeFile. A single LogStore is used to store the updates across all machines. Once the size of the LogStore reaches a threshold, it is compressed into a shard, which spreads over multiple machines. To record the data of each vertex/edge, ZipG uses a set of update pointers, which store offsets in the NodeFiles or EdgeFiles on other machines. The EdgeFile is similar to the adjacency list and used to archive data while the LogStore is similar to the edge list and used to accept updates.

A1 is a distributed in-memory graph storage system developed by Microsoft. As shown in Figure 5(b), A1 jointly utilizes the adjacency list and B-tree. Each edge in A1 keeps edge type, destination vertex address, the size and address of the edge properties. For vertices with small degrees, A1 stores their edges using separate arrays and keeps the edges sorted. For vertices with high degrees (>1000 by default), A1 uses a shared B-tree for the edges of all such vertices, and each edge uses its source vertex and destination vertex as keys. The rationale is to make edge update efficient for high-degree vertices as keeping the edges sorted is expensive (which requires a scan) if the array is large. A1 randomly assigns the vertices to the workers and stores data using FaRM [29], a RDMA-enabled object storage, for efficient communication among the workers.

Terrace uses a 3-level structure as shown in Figure 5(c), i.e., *in-place*, *array*, and *tree*. The place to store a vertex is determined by its degree d_v and two thresholds S and L. The in-place level uses an array, which stores the first S edges of each vertex, and thus vertices with less than S edges only appear in the in-place level. For vertices with $S < d_v \leq S + L$, their additional edges are stored in a PMA shared by all such vertices in the array level. For each vertex with $d_v > S + L$, a separate B-tree is used to store its additional edges. For each vertex, the in-place level keeps a pointer to the position of its edges in the array or tree level.

Terrace is driven by the observation that real graphs usually follow power-law distribution [34], which means that a few vertices have significantly higher degrees than average. By storing the low-degree vertices in an in-place array, they can be fetched together during graph traversal to reduce cache miss. The high-degree vertices are more likely to receive updates, and thus storing them in B-trees benefits from the more efficient update than PMA. Moreover, using separate B-trees for high-degree vertices also allows easy parallelization and reduces locking on PMA. Updates in Terrace follow the standard procedure of the structures. A special case is when the degree of a vertex changes across the thresholds, which requires removing its edges from a level (e.g., PMA) and adding them to another level (e.g., B-tree).

Table 2: Graph datasets used in the experiments

Graph	Vertices	Edges	Type
LiveJournal	4,846,610	68,475,391	Directed
Orkut	3,072,442	234,369,798	Undirected
Uniform-24	13,306,414	520,759,040	Undirected
Graph500-24	16,777,216	520,759,040	Undirected
Twitter	41,652,231	1,468,365,182	Directed
Friendster	68,349,467	2,586,147,869	Directed

4 EXPERIMENTAL EVALUATION

In this part, we conduct extensive experiments to evaluate representative dynamic graph storage systems. We introduce the experiment settings in Section 4.1 and analyze the experiment results for *update*, *read*, *algorithm execution*, *scalability*, and *memory consumption* in Sections 4.2-4.6, respectively. We note that the performance comparisons among the systems are not entirely apple-to-apple as they make different choices such as the data to store and the level of data consistency. In our analysis, we will try to trace the performance influence of the storage structures.

4.1 Experiment Settings

Datasets. We use the graphs in Table 2 for the experiments. Among them, LiveJournal [20], Orkut [4], Twitter [46], and Friendster [1] are real-world graphs widely used to evaluate dynamic graph storage systems. Uniform-24 and Graph500-24 are synthetic graphs generated following [24]. The vertex degrees of Uniform-24 are uniform while Graph500-24 follows the power-law distribution. Due to page limit, we report the results for Orkut (a real and power-law graph) and Uniform-24 (a synthetic and uniform graph, called Uniform for conciseness hereafter) in the paper and provide the results on other graphs in our technical report [6].

Experiment design. We evaluate 9 dynamic graph storage systems, i.e., Teseo [27], Terrace [57], Aspen [28], RisGraph [35], LiveGraph [75], GraphOne [45], LLAMA [51], Stinger [32] and PCSR [71]. We did not evaluate Neo4j [3] and Sparksee [54] as they are designed for disk storage. A1 [22] and Tegra [43] are not included because they are not open-source. For all systems, we adopt the implementations and compilation environments recommended by their papers, and tune the configurable parameters to optimize performance.

We report the performance of the systems in three main aspects, i.e., *update*, *read* and *algorithm*. The design and performance metric of each experiment is introduced as follows:

- *Update*, which inserts/deletes edges for the graph. Following existing works [40, 57], we use the RMAT generator [24] to generate a batch of random edges as the update edges. For each batch, we first insert all update edges into the initial graph and then delete them. We record the time to conduct each batch of edge insertion/deletion and repeat 20 times for each batch size to compute the average time consumption, which is then transformed into update throughput.
- *Read*, which reads local information from a graph and corresponds to lookups. We use three kinds of read operations. *Edge read* checks random edges generated using RMAT, which may or may not exist in the graph, and returns the edge weight if an edge exists. *1-hop* and *2-hop* read the 1-hop and 2-hop neighbors of random vertices, respectively. For each run, we sample 5% of the number of graph edges for edge read and use 5% of the vertices in the graph for 1-hop and 2-hop read. We repeat 20 times and report the average execution time of the read operations.
- *Algorithm*, which runs analytical algorithms on the graph. We use 6 graph algorithms in the LDBC benchmark, i.e., BFS, SSSP, PR, WCC, CDLP, and TC. As discussed in Section 2.1, PR, WCC, and CDLP are global traversal algorithms, BFS and SSSP are local traversal algorithms, and TC conducts fine-grained traversal. For the algorithms, we report the average execution time over 20 runs and exclude the first run to avoid cold cache.

By default, we use 16 threads for the experiments and a batch size of 10^5 for update operations. As PCSR does not support multi-thread update, we use a single thread for it in the update experiments. For the update and read operations, we use the update and read interfaces provided by the systems. In particular, Aspen, Terrace and Stinger provide interface for batch update. To implement the graph algorithms, we adopt the vertex-wise parallel paradigm of Ligra [68], which assigns each thread to handle the workloads on some vertices. Our experiment scripts are open-source [5] with the hope of assisting future researches.

Platform. Our experiments are conducted on a single machine with an Intel Xeon Gold 5122 CPU, which has 8 cores, 16 threads, 16.5MB L3 cache and 512GB DRAM. The Linux kernel version is 4.15.0.

4.2 Update Performance

Figure 6 reports the update performance of the systems. We omit PCSR from the edge deletion experiments as it does not support deleting edges. The results show that the performances for edge insertion and deletion are similar, and the trend are consistent for the two graphs. We make the following observations from Figure 6.

GraphOne, Aspen, PCSR, and Terrace achieve high update throughput among the systems. GraphOne is efficient because it conducts edge updates by appending records to the edge list and does not check the legality of updates (e.g., whether an edge to delete exists). Aspen has high throughput at large batch size because each vertex is more likely to receive multiple updates at large batch size. Aspen first organizes updates for each source vertex as a C-tree and then conducts updates by merging C-trees. PCSR and Terrace adopt sophisticated data structures (i.e., PMA and B-tree), which have a low amortized update cost. Although Teseo also adopts PMA, it has lower throughout than PCSR and Terrace because it conducts legality checks for the updates.

LiveGraph, LLAMA and RisGraph provide low update throughput among the systems. This is partially because they all conduct legality checks for the updates. The edges are not sorted for the three systems, which necessitates scanning all edges of a vertex for legality check and is expensive for the pointer-based structure of LLAMA and RisGraph. LiveGraph stores more information for each edge (i.e., two timestamps and property size), and thus the workload for update is heavy. LLAMA creates a new snapshot for each batch of edge insertion but only marks the deleted edges in their

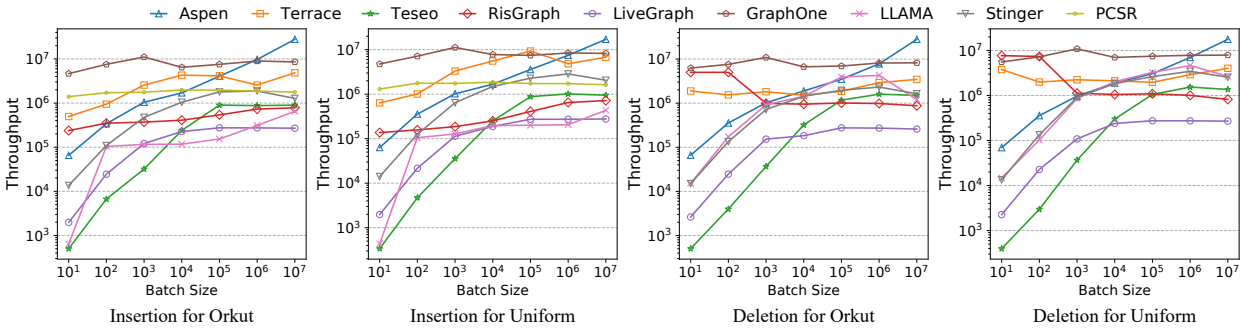


Figure 6: Update throughput on Orkut and Uniform

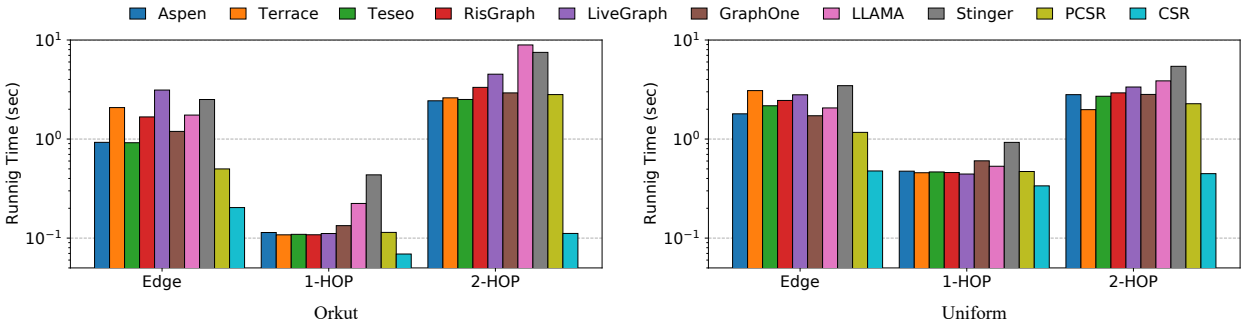


Figure 7: Running time of the read operations on Orkut and Uniform

original snapshots, and thus it achieves much better performance for deletion than insertion.

The update throughput of most systems first increases with batch size but then stabilizes. This is because the relative costs of initializing and scheduling the threads w.r.t. the actual workload are large at small batch size but reduce at large batch size. GraphOne has a stable throughput under different batch sizes as its amortized update cost is low, and thus computation is not the bottleneck. The throughput of Terrace and RisGraph may decrease when increasing the batch size as they are more likely to adjust data structures (e.g., migrating among different levels for Terrace and array resizing for RisGraph) at large batch size.

4.3 Read Performance

We report the running time of the three read operations in Figure 7. CSR, which is used to store static graphs, is included as a baseline. We can make the following observations from Figure 7.

PCSR, Teseo and Aspen achieve good performance for edge read. This is because both PCSR and Teseo adopt the PMA structure to keep the edges of each vertex in a continuous and sorted array, which makes edge lookup fast with binary search. Although Terrace also adopts the PMA structure, it is slower than PCSR and Teseo because edge lookup needs to switch among its 3-level storage structure. Aspen is efficient because its C-tree keeps the edges sorted and delta encoding is utilized to compress the edges, which yields a small memory footprint. LiveGraph and Stinger are slow because their edges are not sorted (which makes edge lookup expensive) and they store additional information with each edge. In particular, LiveGraph keeps two timestamps and a property size for each edge and needs to traverse the edge properties to obtain the edge weight. Stinger keeps two timestamps, a weight and a label for each edge, and its linked list structure also makes edge lookup slow. Although RisGraph builds hash indexes for vertices with more than 512 edges, it is slow as the degrees of most vertices are smaller than 512 for Orkut and Uniform and thus edge lookups are mainly conducted by linear scan.

For 1-hop and 2-hop reads, LLAMA and Stinger are significantly slower while the performance gaps for the other systems are not large. This is because LLAMA and Stinger use linked list and do not store the edges of each vertex continuously. In particular, LLAMA stores the edges of a vertex in different snapshots while Stinger uses blocks in its linked list. LiveGraph becomes more efficient for 1-hop and 2-hop reads than edge read as it stores the edges and their weights separately and read less data for only the graph topology. Overall, the systems have larger performance gap on the power-law Orkut graph than the Uniform graph. This is because Orkut has some large degree vertices, which have a big influence on performance. Note that we conduct experiments for GraphOne on its static view, otherwise, it needs to traverse the entire edge list for each read operation, which is extremely slow.

4.4 Algorithm Execution

Figure 8 reports the execution time of the 6 representative graph analytic algorithms, and the results of CSR are reported as a baseline. As Aspen does not allow edge weight, we exclude it from the experiments for SSSP. We make the following observations.

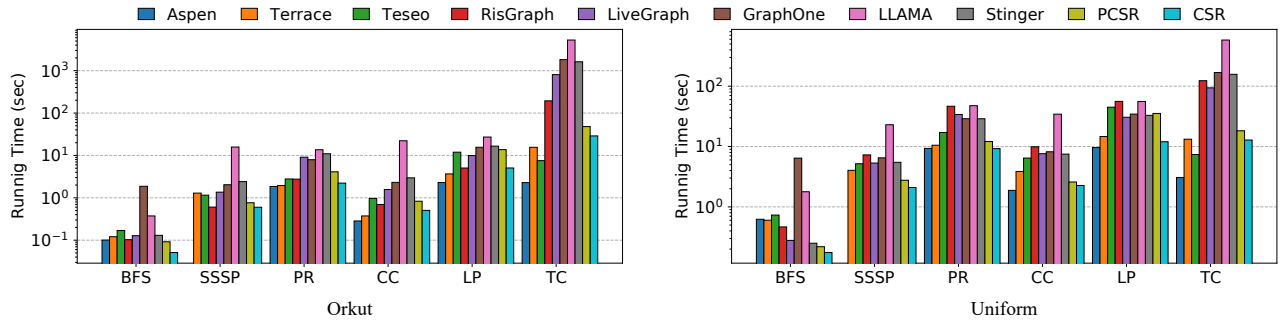


Figure 8: Running time of graph analytics algorithms on Orkut and Uniform

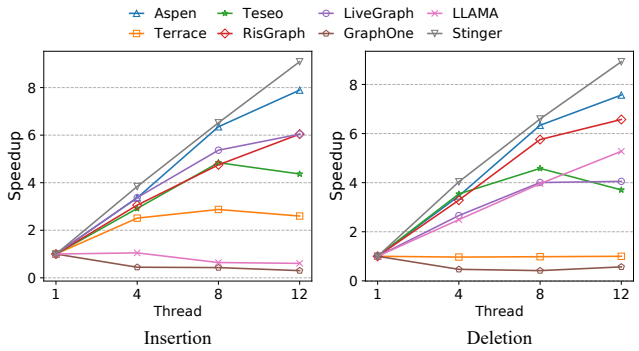


Figure 9: Thread scalability for edge insertion and deletion on Orkut, results for each system normalized by 1 thread

Aspen and systems that adopt an array structure (i.e., Terrace, Teseo, PCSR and RisGraph) are generally fast. This is because Aspen conducts edge compression and does not store edge weight, and thus graph traversal only reads a small amount of memory. The PMA used by Terrace, Teseo, PCSR and the dynamic array of RisGraph keep the edges of each vertex continuous in memory, and thus are also efficient for graph traversal by reducing cache misses. In contrast, LLAMA, Stinger and GraphOne take a long time to run the algorithms because they use linked-list-based storage for the edges of each vertex and thus require expensive pointer chasing. In particular, LLAMA can keep the edges of a vertex in different snapshots and link them by pointers. Stinger and GraphOne directly use linked list of blocks to store the edges. For the TC algorithm, systems that keep the edges sorted (i.e., Terrace, Teseo, PCSR and Aspen) are significantly faster than systems that keep unsorted edges (i.e., RisGraph, LiveGraph, GraphOne, LLAMA, and Stinger).

4.5 Scalability

Thread. Figure 9 reports how the update throughput of the systems scales with the number of threads. This set of experiments are conducted under the default batch size of 10^5, and we did not include PCSR as it does not support multi-thread update. Under each thread count, we normalize the update throughput of a system by its throughput when using 1 thread. This is because the systems have huge differences in actual throughput (see Figure 6), which will make the figure unreadable when plotted together. We only include the results on Orkut due to page limit and note that the results are similar for Uniform (see our technical report). We can make the following observations.

Stinger, Aspen, and RisGraph achieve good scalability for both edge insertion and deletion. Stinger conducts fine-grained locking at edge granularity and thus the updates will not block each other. Aspen and RisGraph use a C-tree [28] and dynamic array for each vertex, respectively, and thus the edges for each vertex can be updated without locking. GraphOne has poor scalability and its update throughput may even drop when using more threads. This is because its edge list is shared by all threads and thus needs to be locked and unlocked frequently. For a similar reason, LLAMA has poor scalability for edge insertion as it stores all new edges in a write store, which is contended by the threads. However, LLAMA has good scalability for edge deletion because the threads can work in parallel to add deletion marks for different edges. Terrace and Teseo have moderate scalability as the PMA structure is shared by all vertices and needs to handle threads contention by locking. In particular, Terrace does not allow multi-thread deletion (which explains the horizontal scalability curve), and Teseo locks a leaf of the fat-tree for re-balance. The update throughput of Teseo drops when increasing from 8 threads to 12 threads because it spawns a service thread on each core to conduct re-balancing and garbage collection, and using 12 threads for update leaves only 4 service threads for the 8 cores. The drop of scalability for LiveGraph is also observed by [27], and may be caused by its transaction manager.

Figure 10 reports the running time of the systems for the read workloads and graph analytics algorithms when using the different number of threads. Due to page limit, we select BFS and PR as representatives of the algorithms, and only include the results on Orkut. The results for the other algorithms and the Uniform graph exhibit similar trends, and can be found in our technical report. Figure 10 shows that the systems generally achieve good thread scalability for the read operations and analytical algorithms as these tasks only read data and have no contention among the threads. The exception is for 1-hop read, and the poor scalability is because the workload is lightweight and thus the overheads of launching and scheduling the threads are substantial. All systems have better scalability for PR than for BFS. This is because BFS only accesses the edges of some vertices in each iteration while PR accesses all edges in the graph, and thus PR can better utilize the threads.

Data. Figure 11 reports the running time of the read operations and analytics algorithms on graphs with different sizes when using 16

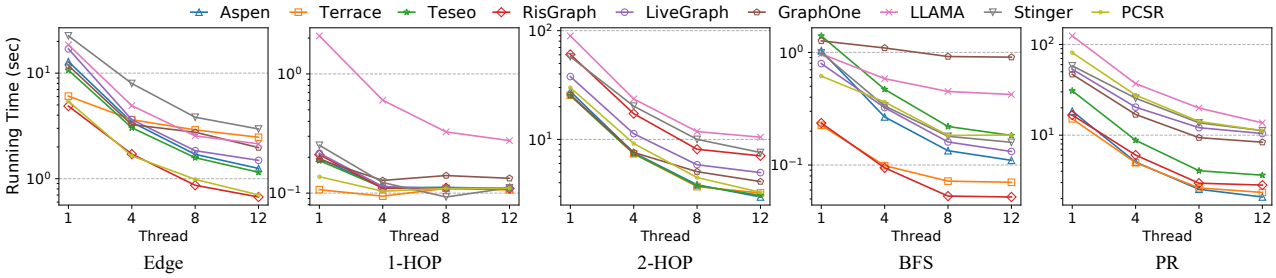


Figure 10: Thread scalability for the read operations and graph analytics algorithms on Orkut

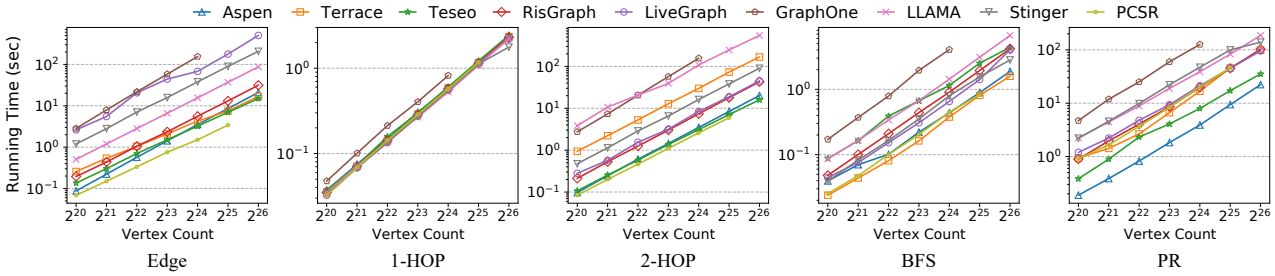


Figure 11: Data scalability on the synthetic and power-law Graph500 graph by adjusting vertex count

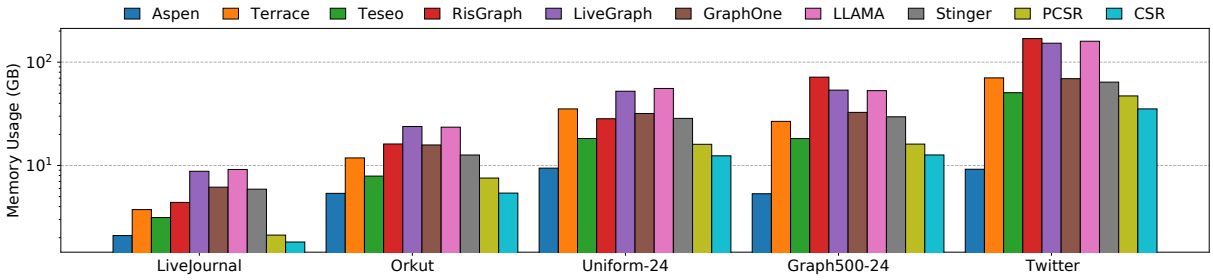


Figure 12: Memory consumption of the systems for storing the experiment graph

threads. We use RMAT to generate a power-law graph and control the number of vertices by keeping an average degree of 30. Some data points are missing because some systems cannot load the graphs. The results show that the running time scales linearly with the size of the graph (note that both axes use a log scale). We also tried to fix the number of vertices and change the average vertex degree, and observe that the running time scales linearly with the number of edges in the graph. The update throughput of the systems generally exhibits a slowly decreasing trend w.r.t. graph size. Due to page limit, we refer interested readers to our technical report for these results.

4.6 Memory Consumption

Figure 12 reports the memory costs of the systems for storing the experiment graphs, and CSR is included as a baseline. The results show that Aspen consumes the least memory in most cases and the advantage is more significant for large graphs. This is because Aspen only stores the graph topology and compresses the edges. In contrast, LiveGraph store additional timestamps and property size for each edge, and thus have large memory consumption. LLAMA consumes a large amount of memory because it uses snapshots and each snapshot needs to copy the indirection array and vertex array even if there are only a few updates. The memory consumption of RisGraph is low for small graphs but large for big graphs. This is because RisGraph builds a hash index for vertices with more than 512 edges and large graphs have more high degree vertices. Teseo, PCSR and Terrace reserve space for efficient updates and thus have larger memory consumption than CSR. Although the three systems all use PMA, Terrace consumes more memory as its in-place array has a small fill factor.

5 DISCUSSIONS

As shown by our survey in Section 3 and experiments in Section 4, existing dynamic graph storage systems are designed with different considerations, and there is no single winner that dominates the others in all aspects. We also find that existing systems boil down to a set of techniques that can be composed to meet different application requirements. In this part, we summarize the key design choices our should make when designing a storage system for dynamic graph and highlight some key techniques.

Structure, store the edges of each vertex in a separate structure (e.g., the C-tree in Aspen and dynamic array in RisGraph) or store

the edges of multiple vertices in a shared structure (e.g., the PMA in Teseo)? Independent structure simplifies parallel execution, especially for updates, and results in good thread scalability. However, if the graph contains many low-degree vertices, the independent structure may hinder algorithm execution as the edges for vertex cannot fill a cache line. The shared structure requires locking for updates (e.g., re-balance for PMA) and fine-grained locking (e.g., the edge-level locking in Stinger) is important for thread scalability. By reading the edges of multiple vertices to the cache in one pass, shared structure benefits global traversal algorithms, especially when there are many low-degree vertices.

Property, does the application need to store additional properties for each vertex/edge, and how to store the properties? As shown by LiveGraph and Stinger, storing properties (e.g., timestamps and property size) is necessary for certain purposes (e.g., version tracking and concurrency control) but degrades performance in all aspects. In contrast, Aspen stores only the graph topology and performs well in several aspects. We think the properties need to be stored according to their access patterns. Frequently accessed properties should be stored in-line along with each vertex/edge such that their accesses are piggybacked with the graph topology (e.g., the edge weight when running SSSP). However, properties that are accessed infrequently should be stored in separate structures to reduce the memory footprint for update and read operations.

Legality, does the data source ensure the legality of updates (e.g., no duplicate edges)? If not, the system needs to conduct legality check for updates, structures for edge lookup (e.g., PMA in Teseo and hash index in RisGraph) should be used to quickly locate the edges. These structures also accelerate fine-grained traversal algorithms (i.e., TC) by pinpointing certain edges without scan. Other indexes can also be tailored according to the access pattern of the application (e.g., the pART for range scan in Tegra). The index can be either the storage structure itself by maintaining certain invariants (e.g., PMA) or separate from the main storage structure (e.g., the hash index in RisGraph and the ETA in Stinger).

Multiple structures, which jointly utilize multiple structures as different structures excel in different aspects. PMA is an efficient shared structure to maintain sorted edges and has only moderate memory overhead (the fill factor is usually above 0.75 for Teseo and Terrace). With block size aligned to the cache line, the tree structures (e.g., B-tree and C-tree) have low space overhead and are more efficient for update but slower for graph traversal than PMA. Edge list excels in update performance and allows fine-grained version tracking but is slow for traversal. For example, Terrace uses a three-level architecture with array, PMA and B-tree for vertices having different degrees. We think this methodology is a promising direction for future research by developing other combinations of basic structures, and one may want to consider it when designing graph storage.

Compression, for example, the delta encoding used by Aspen to compress the edges of each vertex. Compression reduces memory consumption and accelerates graph analytics algorithms by reading less data. Beside the edges, we think compression may also be applied to other data such as weights and timestamps. It is important that the underlying data layout is suitable for compression (e.g., delta encoding in Aspen is enabled by sorted edges) and to control the costs of compression and decompression.

Batch optimization, which groups the updates according to their target vertex as in Aspen and Terrace. This allows to amortize the update cost and achieve high update throughput at large batch size.

6 CONCLUSIONS AND FUTURE DIRECTIONS

In this paper, we conduct an extensive survey of existing dynamic graph storage systems and evaluate their performance with extensive experiments. We find that there is no single winner in existing systems that dominate the others in all aspects, and different systems make different trade-offs in their designs. To facilitate the design of future graph storage systems, we characterize the performance implications of different storage structures, summarize the key design requirements one should consider, and discuss how existing techniques can be utilized to meet these requirements. We think there are several promising directions for future research.

Adaptive storage structure. There is a conflict between traversal and update performance for existing storage structures. Specifically, a compact array (as in CSR) is the most efficient for traversal but expensive for update; PMA makes update efficient by reserving space but scarifies some traversal performance; pointer-based structures (e.g., B-tree and C-tree) can be regarded as reserving infinite space between data, and are efficient for update but slow for traversal due to pointer chasing. Terrace observes that high degree vertices are more likely to receive updates and store nodes with larger degrees in structures with more reserved spaces. However, the transition among array, PMA and trees is discrete based on degree thresholds. A better trade-off between traversal and update performance may be achieved by designing an adaptive data structure that smoothly adjusts the reserved space for nodes with different degrees.

Multi-view storage. Different applications may run on the same graph, and it can be difficult to design a graph storage to suit all applications as their requirements may conflict with each other. For example, BFS and TC only access the graph topology and are efficient if edge weight is stored in separate structures. However, SSSP relies on edge weight and requires to store edge weight in-line with the graph topology. The situation becomes more complex when each vertex/edge contains more properties and different applications access different properties. It is possible to maintain different storage views of the same graph, with each view tailored for one class of applications/queries, in a spirit similar to materialized views in databases. Mechanisms need to be designed to ensure the consistency among the views and control memory consumption.

Large-scale solutions. Existing systems mainly consider the main memory for a single machine, which may not be able to hold extremely large graphs. One option is to scale up the capacity of one machine with SSD. Although the bandwidth of current SSDs can reach more than 2GB/s, their latency is still more than 100x of DRAM. Moreover, designs should also consider that SSD accesses are conducted in 4KB blocks. The other option is scale out with multiple machines. RDMA may be utilized for efficient cross-machine communication as in A1 [22] but more sophisticated designs (e.g., using only one-sided verbs) may be required to reap the full benefits of RDMA.

REFERENCES

[1] 2022. *Friendster Network Dataset – KONECT.* http://konect.cc/networks/friendster/

[2] 2022. *Giraph.* http://giraph.apache.org/

[3] 2022. *neo4j.* https://neo4j.com/

[4] 2022. *Orkut Network Dataset – KONECT.* http://konect.cc/networks/orkut-links/

[5] 2022. *Survey experiment implementation source code.* https://github.com/xiangyuzhi/GraphStorageExp/

[6] 2022. *Technical report of dynamic graph storage.* https://drive.google.com/file/d/1scI0sU83y-vcPnT5Vmjg4tif_wMlGj20/view?usp=sharing

[7] Umut A Acar, Daniel Anderson, Guy E Blelloch, and Laxman Dhulipala. 2019. Parallel batch-dynamic graph connectivity. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*. 381–392.

[8] Umut A Acar, Andrew Cotter, Benoît Hudson, and Duru Türkoglu. 2011. Parallelism in dynamic well-spaced point sets. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*. 33–42.

[9] Rachit Agarwal, Anurag Khandelwal, and Ion Stoica. 2015. Succinct: Enabling queries on compressed data. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. 337–350.

[10] Leman Akoglu, Hanghang Tong, and Danai Koutra. 2015. Graph based anomaly detection and description: a survey. *Data mining and knowledge discovery* 29, 3 (2015), 626–688.

[11] Khaled Ammar and Tamer Ozsu. 2018. Experimental analysis of distributed graph systems. *Proceedings of the VLDB Endowment* 11, 10 (2018), 1151–1164.

[12] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. 2017. Foundations of modern query languages for graph databases. *ACM Computing Surveys (CSUR)* 50, 5 (2017), 1–40.

[13] Timothy G Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. 2013. Linkbench: a database benchmark based on the facebook social graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 1185–1196.

[14] Michael A Bender, Erik D Demaine, and Martin Farach-Colton. 2000. Cache-oblivious B-trees. In *Proceedings 41st Annual Symposium on Foundations of Computer Science*. IEEE, 399–409.

[15] Michael A Bender and Haodong Hu. 2007. An adaptive packed-memory array. *ACM Transactions on Database Systems (TODS)* 32, 4 (2007), 26–es.

[16] Maciej Besta, Marc Fischer, Vasiliki Kalavri, Michael Kapralov, and Torsten Hoefler. 2021. Practice of streaming processing of dynamic graphs: Concepts, models, and systems. *IEEE Transactions on Parallel and Distributed Systems* (2021).

[17] Maciej Besta, Emanuel Peter, Robert Gerstenberger, Marc Fischer, Michał Podstawski, Claude Barthels, Gustavo Alonso, and Torsten Hoefler. 2019. Demystifying graph databases: Analysis and taxonomy of data organization, system designs, and graph queries. *arXiv preprint arXiv:1910.09017* (2019).

[18] Alex Beutel, Leman Akoglu, and Christos Faloutsos. 2015. Fraud detection through graph-based user behavior modeling. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 1696–1697.

[19] Sayan Bhattacharya, Monika Henzinger, and Danupon Nanongkai. 2019. A new deterministic algorithm for dynamic set cover. In *2019 IEEE 60th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 406–423.

[20] Paolo Boldi and Sebastiano Vigna. 2004. The webgraph framework I: compression techniques. In *Proceedings of the 13th international conference on World Wide Web*. 595–602.

[21] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, et al. 2013. TAO:Facebook's Distributed Data Store for the Social Graph. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. 49–60.

[22] Chiranjeeb Buragohain, Knut Magne Risvik, Paul Brett, Miguel Castro, Wonhee Cho, Joshua Cowhig, Nikolas Gloy, Karthik Kalyanaraman, Richendra Khanna, John Pao, et al. 2020. A1: A distributed in-memory graph database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 329–344.

[23] Yukuo Cen, Jing Zhang, Gaofei Wang, Yujie Qian, Chuizheng Meng, Zonghong Dai, Hongxia Yang, and Jie Tang. 2019. Trust relationship prediction in alibaba E-commerce platform. *IEEE Transactions on Knowledge and Data Engineering* 32, 5 (2019), 1024–1035.

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

[25] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. 2015. One trillion edges: Graph processing at facebook-scale. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1804–1815.

[26] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2022. *Introduction to algorithms*. MIT press.

[27] Dean De Leo and Peter Boncz. 2021. Teseo and the analysis of structural dynamic graphs. *Proceedings of the VLDB Endowment* 14, 6 (2021), 1053–1066.

[28] 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*. 918–934.

[29] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. 401–414.

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

[31] David Easley and Jon Kleinberg. 2010. *Networks, crowds, and markets: Reasoning about a highly connected world*. Cambridge university press.

[32] David Ediger, Rob McColl, Jason Riedy, and David A Bader. 2012. Stinger: High performance data structure for streaming graphs. In *2012 IEEE Conference on High Performance Extreme Computing*. IEEE, 1–5.

[33] Stephen Eubank, Hasan Guclu, VS Anil Kumar, Madhav V Marathe, Aravind Srinivasan, Zoltan Toroczkai, and Nan Wang. 2004. Modelling disease outbreaks in realistic urban social networks. *Nature* 429, 6988 (2004), 180–184.

[34] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. 1999. On power-law relationships of the internet topology. *ACM SIGCOMM computer communication review* 29, 4 (1999), 251–262.

[35] Guanyu Feng, Zixuan Ma, Daixuan Li, Shengqi Chen, Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2021. RisGraph: A Real-Time Streaming System for Evolving Graphs to Support Sub-millisecond Per-update Analysis at Millions Ops/s. In *Proceedings of the 2021 ACM SIGMOD International Conference on Management of Data*. 513–527.

[36] Soukaina Firmli, Vasileios Trigonakis, Jean-Pierre Lozi, Iraklis Psaroudakis, Alexander Weld, Dalila Chiadmi, Sungpack Hong, and Hassan Chafi. 2020. CSR++: A Fast, Scalable, Update-Friendly Graph Data Structure. In *24th International Conference on Principles of Distributed Systems (OPODIS'20)*.

[37] Sanchit Garg, Trinabh Gupta, Niklas Carlsson, and Anirban Mahanti. 2009. Evolution of an online social aggregation network: an empirical study. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement*. 315–321.

[38] Pranjal Gupta, Amine Mhedhbi, and Semih Salihoglu. 2021. Columnar storage and list-based processing for graph database management systems. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2491–2504.

[39] Bernardo A Huberman and Lada A Adamic. 1999. Growth dynamics of the world-wide web. *Nature* 401, 6749 (1999), 131–131.

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

[41] Alon Itai, Alan G Konheim, and Michael Rodeh. 1981. A sparse table implementation of priority queues. In *International Colloquium on Automata, Languages, and Programming*. Springer, 417–431.

[42] Anand Padmanabha Iyer, Li Erran Li, Tathagata Das, and Ion Stoica. 2016. Time-evolving graph processing at scale. In *Proceedings of the fourth international workshop on graph data management experiences and systems*. 1–6.

[43] Anand Padmanabha Iyer, Qifan Pu, Kishan Patel, Joseph E Gonzalez, and Ion Stoica. 2021. TEGRA: Efficient Ad-Hoc Analytics on Evolving Graphs. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 337–355.

[44] Anurag Khandelwal, Zongheng Yang, Evan Ye, Rachit Agarwal, and Ion Stoica. 2017. Zipg: A memory-efficient graph store for interactive queries. In *Proceedings of the 2017 ACM SIGMOD International Conference on Management of Data*. 1149–1164.

[45] Pradeep Kumar and H Howie Huang. 2020. Graphone: A data store for real-time analytics on evolving graphs. *ACM Transactions on Storage (TOS)* 15, 4 (2020), 1–40.

[46] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media?. In *Proceedings of the 19th international conference on World wide web*. 591–600.

[47] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 38–49.

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

[49] Matteo Lissandrini, Martin Brugnara, and Yannis Velegrakis. 2018. Beyond macrobenchmarks: microbenchmark-based graph database evaluation. *Proceedings of the VLDB Endowment* 12, 4 (2018), 390–403.

[50] Yi Lu, James Cheng, Da Yan, and Huanhuan Wu. 2014. Large-scale distributed graph computing systems: An experimental evaluation. *Proceedings of the VLDB Endowment* 8, 3 (2014), 281–292.

[51] Peter Macko, Virendra J Marathe, Daniel W Margo, and Margo I Seltzer. 2015. Llama: Efficient graph analytics using large multiversioned arrays. In *2015 IEEE 31st International Conference on Data Engineering*. IEEE, 363–374.

[52] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 135–146.

[53] Renxin Mao, Zhao Li, and Jinhua Fu. 2015. Fraud transaction recognition: A money flow network approach. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*. 1871–1874.

[54] Norbert Martínez-Bazan, M Ángel Águila-Lorente, Victor Muntés-Mulero, David Dominguez-Sal, Sergio Gómez-Villamor, and Josep-L Larriba-Pey. 2012. Efficient graph management based on bitmap indices. In *Proceedings of the 16th International Database Engineering & Applications Sysmposium*. 110–119.

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

[56] Krzysztof Nowicki and Krzysztof Onak. 2021. Dynamic graph algorithms with batch updates in the massively parallel computation model. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM, 2939–2958.

[57] Prashant Pandey, Brian Wheatman, Helen Xu, and Aydin Buluc. 2021. Terrace: A hierarchical graph container for skewed dynamic graphs. In *Proceedings of the 2021 ACM SIGMOD International Conference on Management of Data*. 1372–1385.

[58] Vijayan Prabhakaran, Ming Wu, Xuetian Weng, Frank McSherry, Lidong Zhou, and Maya Haradasan. 2012. Managing Large Graphs on Multi-Cores with Graph Awareness. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. 41–52.

[59] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. 2018. Real-time constrained cycle detection in large dynamic graphs. *Proceedings of the VLDB Endowment* 11, 12 (2018), 1876–1888.

[60] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. 2007. Near linear time algorithm to detect community structures in large-scale networks. *Physical review E* 76, 3 (2007), 036106.

[61] Ian Robinson, Jim Webber, and Emil Eifrem. 2015. *Graph databases: new opportunities for connected data.* " O'Reilly Media, Inc.".

[62] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. 2015. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the 25th Symposium on Operating Systems Principles*. 410–424.

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

[64] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M Tamer Özsu. 2020. The ubiquity of large graphs and surprising challenges of graph processing: extended survey. *The VLDB journal* 29, 2 (2020), 595–618.

[65] John Scott. 1988. Social network analysis. *Sociology* 22, 1 (1988), 109–127.

[66] Mo Sha, Yuchen Li, Bingsheng He, and Kian-Lee Tan. 2017. Accelerating Dynamic Graph Analytics on GPUs. *Proceedings of the VLDB Endowment* 11, 1 (2017), 107–120.

[67] Feng Sheng, Qiang Cao, and Jie Yao. 2020. Exploiting buffered updates for fast streaming graph analysis. *IEEE Trans. Comput.* 70, 2 (2020), 255–269.

[68] 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*. 135–146.

[69] Julian Shun, Laxman Dhulipala, and Guy E Blelloch. 2015. Smaller and faster: Parallel processing of compressed graphs with Ligra+. In *2015 Data Compression Conference*. IEEE, 403–412.

[70] Keval Vora, Rajiv Gupta, and Guoqing Xu. 2017. Kickstarter: Fast and accurate computations on streaming graphs via trimmed approximations. In *Proceedings of the twenty-second international conference on architectural support for programming languages and operating systems*. 237–251.

[71] Brian Wheatman and Helen Xu. 2018. Packed compressed sparse row: A dynamic graph representation. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 1–7.

[72] Wenpu Xing and Ali Ghorbani. 2004. Weighted pagerank algorithm. In *Proceedings. Second Annual Conference on Communication Networks and Services Research, 2004.* IEEE, 305–314.

[73] Lei Yang, Lei Qi, Yan-Ping Zhao, Bin Gao, and Tie-Yan Liu. 2007. Link analysis using time series of web graphs. In *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*. 1011–1014.

[74] 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*. 614–630.

[75] Xiaowei Zhu, Guanyu Feng, Marco Serafini, Xiaosong Ma, Jiping Yu, Lei Xie, Ashraf Aboulnaga, and Wenguang Chen. 2020. LiveGraph: A transactional graph storage system with purely sequential adjacency list scans. *Proceedings of the VLDB Endowment* 13, 7 (2020), 1020–1034.

[76] Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. GridGraph:Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 375–386.