Bw-Graph: An Efficient Graph Storage System Harmonizing Topology-Aware Tree with Paged CSR

SONGYAO WANG, Tsinghua University, China
CHAOKUN WANG*, Tsinghua University, China
ZECHENG LI, Tsinghua University, China
AOQI ZHANG, Tsinghua University, China

As digital transformation accelerates, modern graph datasets routinely scale to billions of vertices and edges, demanding storage systems that simultaneously support dynamic updates, rapid neighbor retrieval, and high-performance analytics. However, existing systems face significant performance bottlenecks. While LSM-Tree-based approaches have gained popularity for their write efficiency, they suffer from fundamental read amplification problems that remain unresolved despite various optimization attempts. Moreover, most systems organize data by vertex IDs for simplicity, thereby sacrificing topological locality that could benefit graph algorithms. Although recent topology-aware methods offer improvements, they suffer from severe partition imbalance and fragility under updates. These challenges are further compounded by concurrency control designs in which structural modification operations block user requests, thereby limiting system throughput.

To address these challenges, we present Bw-Graph, a graph storage system that harmonizes a Topology-Aware Tree with Paged CSR. Bw-Graph employs CSR pages for efficient neighbor access, while utilizing append-only Δ Pages to enable sequential writes to subgraphs. To enable efficient graph analytics, we propose a Topology-Aware Tree that hierarchically organizes graph data to co-locate densely connected vertices in contiguous physical storage. The underlying weight-constrained graph partitioning scheme ensures balanced partition sizes while preserving topological locality. To support high concurrency, we develop a tailored MVCC mechanism that leverages the append-only nature of Δ Pages for lightweight version management, and exploits a Multi-Version Vertex Index to guarantee consistency without blocking user requests during structural modification operations. Comprehensive evaluations demonstrate that Bw-Graph achieves significant performance improvements across diverse workloads. For analytics tasks, Bw-Graph further delivers performance comparable to that of dedicated graph processing systems (e.g., GridGraph).

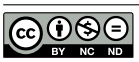
CCS Concepts: • **Information systems** → *Graph-based database models*; **Data structures**.

Additional Key Words and Phrases: Graph Storage System, Topology-Aware Organization, Bw-Tree, CSR Representation

ACM Reference Format:
Songyao Wang, Chaokun Wang, Zecheng Li, and Aoqi Zhang. 2026. Bw-Graph: An Efficient Graph Storage System Harmonizing Topology-Aware Tree with Paged CSR. *Proc. ACM Manag. Data* 4, 3 (SIGMOD), Article 148 (June 2026), 27 pages. https://doi.org/10.1145/3802025

*Corresponding author (chaokun@tsinghua.edu.cn).

Authors' Contact Information: Songyao Wang, Tsinghua University, Beijing, China, wangsong23@mails.tsinghua.edu.cn; Chaokun Wang, Tsinghua University, Beijing, China, chaokun@tsinghua.edu.cn; Zecheng Li, Tsinghua University, Beijing, China, zc-li23@mails.tsinghua.edu.cn; Aoqi Zhang, Tsinghua University, Beijing, China, zaq23@mails.tsinghua.edu.cn.



ACM 2836-6573/2026/6-ART148
https://doi.org/10.1145/3802025

1 Introduction

Graphs have become essential data structures for modeling complex relationships in the modern interconnected world. Unlike traditional data structures, graphs naturally capture the connections between real-world entities, such as social networks among users, interactions between proteins in biological systems, transportation routes in urban infrastructures, and semantic links in knowledge bases. This ability to represent both entities and their relationships makes graphs particularly suitable for applications where the connectivity is central to understanding the underlying system. Graph algorithms, including PageRank and community detection, enable researchers to systematically analyze structural properties and interaction patterns within these networks. As data generation accelerates across domains, from social media platforms to scientific collaborations, the scale of graph data continues to grow at an unprecedented rate. As a result, modern graph datasets routinely contain millions to billions of vertices and edges. For instance, Web graphs such as IT-2004 [16] and UK-2007 [2] contain over 100 million pages and billions of links, while social networks like X (Twitter) [46] exceed 500 million users and tens of billions of connections. This unprecedented growth poses significant challenges in efficient storage, processing, and real-time analysis of graph data.

To address these scalability challenges, graph storage systems have become fundamental infrastructure for enabling efficient analytics on massive datasets. These systems must persistently store large-scale data, handle dynamic updates, and provide efficient support for query processing and analytical algorithms. For instance, social networking platforms such as Meta (Facebook) and X (Twitter) store billions of users and hundreds of billions of relationships, while continuously processing a high volume of updates from new connections and evolving interactions. To support analytical tasks such as user behavior studies and influence propagation, these systems must efficiently execute community detection algorithms [9] and pattern matching queries [29, 30] to identify frequently communicating user groups within large-scale social networks. Furthermore, modern storage systems are increasingly required to support deep learning workloads [26], such as graph sampling and coarsening operations [27] in graph contrastive learning, which demand efficient access to topological structures.

From the perspective of efficient graph query processing, an ideal graph storage system should possess three key capabilities. ① ***Handle graph updates efficiently***. Real-world graphs are inherently dynamic, with relationships and entities constantly evolving. Moreover, these updates often exhibit strong locality, as operations like vertex deletion and neighbor property propagation naturally trigger cascading modifications across topologically adjacent vertices within localized subgraph regions. This capability requires processing vertex and edge operations (e.g., insertion, deletion, and modification) with low latency, ideally by exploiting such update locality to enable sequential subgraph writes rather than scattered random modifications. ② ***Scan neighbors rapidly***. Neighbor retrieval serves as a fundamental operator in graph algorithms [39]. For instance, computing the shortest path between two vertices requires repeatedly accessing the neighbors of intermediate vertices during traversal. Therefore, reducing neighbor scan latency can significantly improve overall system performance [39]. ③ ***Support high-performance graph analytics***. Modern graph applications demand sophisticated analytical tasks that involve intensive computational workloads. For example, community detection algorithms [5, 24, 44, 45, 53] must simultaneously process millions of vertices and edges to identify densely connected subgraphs within large-scale networks. In summary, efficient update and neighbor scan form the foundation for graph query processing, while high-performance analytics enables complex computational workloads [14].

The Challenge. Achieving all three capabilities simultaneously, however, remains a formidable challenge. Existing graph storage systems are fundamentally constrained by conflicting design

Table 1. The time complexity comparison of data structures. N represents the total number of entries ($|V|$ in graph context), F denotes the fanout or growth factor, δ is the length of delta chain or Δ Page chain, and L is the number of levels in LSM-Tree. SMO overheads are not considered in this table.

Data Structure	Read Disk I/O	Read Memory Access	Write Disk I/O	Write Memory Access	Read Efficiency	Write Efficiency
CSR	$O(1)$	$O(1)$	$O(N)$	$O(N)$	High	Low
B^+-Tree	$O(\log_F N)$	$O(\log_2 F)$	$O(\log_F N)$	$O(F)$	High	Low
LSM-Tree	$O(L \cdot \log_F N)$	$O(L \cdot \log_2 F)$	$O(1)$	$O(\log_2 F)$	Low	High
Bw-Tree	$O(\log_F N)$	$O(\log_F N + \delta)$	$O(\delta)$	$O(1)$	Med.	High
Bw-Graph	$O(1+\delta)$	$O(1+\delta)$	$O(\delta)$	$O(1)$	High	High

trade-offs between high-throughput writes and low-latency reads. Specifically, write-optimized systems [39, 57] suffer from read amplification during neighbor scans, while read-optimized designs [34, 61] impose prohibitive costs for updates. Analytics-oriented systems [21, 59, 62] deliver strong analytical performance but either do not support dynamic updates [21, 62] or require expensive graph reorganization upon updates [59], making them unsuitable for transactional workloads, which involve frequent updates and low-latency read operations. Even recent HTAP approaches [14] that attempt to reconcile transactional and analytical processing remain constrained by ID-based data organization, which ignores topological locality and leaves substantial performance gains untapped. These fundamental limitations motivate us to rethink the design choices in graph storage, revealing the following optimization opportunities.

O1. Select an appropriate "storage infrastructure". Graph storage systems build upon and adapt fundamental data structures to better serve graph workloads. In recent years, LSM-Trees [42] have gained popularity in graph storage [14, 39, 57]. For example, LSMGraph [57] extensively redesigns the LSM-Tree architecture with write-optimized mechanisms to amortize the update costs of read-optimized CSR, achieving both efficient neighbor retrieval and data ingestion. Beyond LSM-Trees, various systems [6, 34, 58, 61] leverage alternative data structures, such as CSR, adjacency lists, B^+-Trees, and Bw-Trees. For instance, LLAMA [34] employs a log-based approach to stack CSR snapshots, achieving reasonable read performance at the cost of increased storage overhead [57]. While LSM-Tree-based systems have demonstrated impressive performance gains, they inherit the fundamental read amplification problem inherent to LSM-Trees. Although these systems adopt optimization strategies, e.g., adaptive update strategy selection [39] and CSR integration [14, 39], to mitigate read amplification, the read amplification problem intrinsic to LSM-Trees remains fundamentally unresolved. To fundamentally address read amplification, it is essential to draw inspiration from more suitable "storage infrastructures", i.e., leveraging design principles from next-generation data structures (e.g., ***Bw-Tree*** in Table 1) to architect novel storage engines with superior read-write trade-offs.

O2. Organize graph data with topology information rather than IDs. While most graph storage systems favor vertex identifiers (IDs) for simplicity, recent studies [54, 59] show that topology-aware organization (e.g., using SCCs or communities) significantly enhances algorithm efficiency through better locality and pruning. However, existing approaches like DGraph [59] are fragile under updates due to SCC instability, and suffer from severe skewness [54] (e.g., a single giant component). Beyond algorithmic benefits, topology-aware organization fundamentally optimizes write I/O patterns by co-locating topologically related vertices. This co-location of topologically adjacent vertices enables sequential writes to subgraphs rather than scattered updates throughout the ID space. The locality advantage persists even in LSM-tree-based systems. While LSM-trees achieve sequential writes globally, scattered updates spanning large ID ranges create wide-ranging

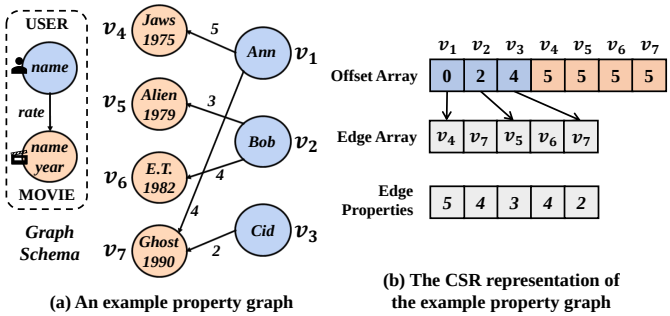


Fig. 1. An example property graph and its CSR representation. (a) The property graph alongside its schema. (b) The corresponding CSR representation.

SSTables that amplify compaction overhead by overlapping numerous lower-level files, indirectly reintroducing write amplification. Consequently, there is a compelling need to develop novel topology-aware organization approaches that avoid skewness and instability while unlocking write efficiency gains.

O3. Employ vertex-centric MVCC to mitigate read-write conflicts. Existing graph systems typically employ sophisticated concurrency control protocols to ensure consistency across read, write, and structural modification operations (SMOs, e.g., page splits and merges in B^+-Trees, and compactions in LSM-Trees). For instance, LSMGraph [57] implements a lightweight vertex-granularity concurrency control protocol to guarantee that each read operation accesses the correct version of data. However, the long-held vertex-level locks during SMOs inevitably block concurrent read threads, leading to performance degradation. Therefore, it is essential to design a novel vertex-centric MVCC mechanism that ensures consistency while enabling high-performance concurrent operations.

Our Approach. Motivated by the optimization opportunities discussed above, we propose Bw-Graph, a novel graph storage system inspired by Bw-Tree's latch-free and log-structured design principles [25], which harmonizes Topology-Aware Tree with Paged CSR (§ 3). The key innovation is a ***Topology-Aware Tree (TAT)***, an index structure that organizes vertices by topological proximity rather than logical identifiers (§ 4). This organization ensures that densely connected vertices tend to be co-located in contiguous, CSR-formatted base pages (CSR pages), enabling efficient neighbor access while maintaining tree-based indexing benefits. Leveraging this topology-aware organization, Bw-Graph employs ***delta page (Δ Page) chains*** to convert scattered vertex and edge updates into ***sequential writes to subgraphs***, enabling high-throughput concurrent graph updates. To preserve this locality under updates, we design specialized SMOs that enable lock-free delta updates on CSR-structured pages while maintaining both compactness and topological locality (§ 5). Finally, we develop a tailored MVCC protocol to ensure snapshot consistency for analytics queries over the dynamically evolving TAT (§ 6).

Contributions. Our main contributions are summarized as follows:

(1) We propose Bw-Graph, a graph storage system that harmonizes Topology-Aware Tree with Paged CSR, employing CSR pages for efficient neighbor scan while leveraging Δ Page chains for sequential writes to subgraphs.
(2) We design a Topology-Aware Tree that hierarchically organizes graph data to co-locate densely connected vertices in contiguous physical storage, exploiting topological locality to accelerate graph analytics.

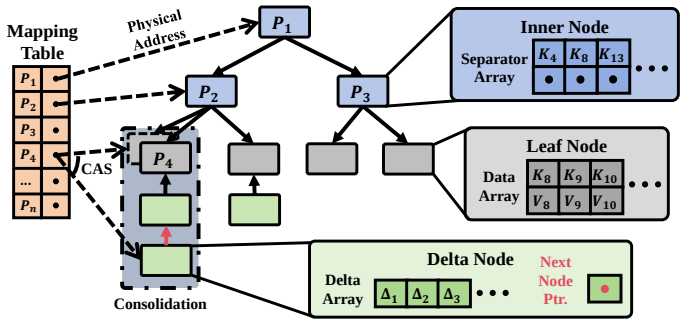


Fig. 2. The architecture of Bw-Tree. It highlights the mapping table for logical-to-physical address translation, along with inner, leaf, and delta nodes.

(3) We present a tailored MVCC mechanism inspired by Bw-Tree's latch-free and log-structured design principles that enables efficient multi-version management with high concurrent performance.
(4) We conduct comprehensive experiments demonstrating that Bw-Graph achieves significant performance improvements over existing graph storage systems while remaining competitive with dedicated graph processing systems on analytical workloads.

2 Background

This section provides the necessary background on the CSR representation for property graphs and reviews the Bw-Tree, a widely adopted lock-free indexing structure.

Compressed Sparse Row (CSR) is a space-efficient and read-friendly format that encodes graph topology using contiguous memory. As illustrated in Figure 1, it maintains an *offset array* to index the starting position of each vertex's neighbor list within a flattened *edge array*. For example, to identify the movies rated by user *Bob* (v_2), the offset at Position 2 of the offset array points to the contiguous segment in the edge array containing $\{v_5, v_6\}$. This compact layout enables $O(1)$ access to vertex neighbors and maximizes cache locality. However, structural updates are costly, requiring significant data movement proportional to the graph size to maintain the contiguity. Consequently, in practice, CSR is primarily used as an immutable structure [14, 57] for read-heavy workloads, analogous to Sorted String Tables (SSTs) [42] in LSM-Tree-based storage engines in terms of immutability and read optimization.

Bw-Tree [25] is a latch-free, log-structured B-tree variant designed for high-concurrency environments by employing compare-and-swap (CAS) operations on page pointers rather than locks or latches. As illustrated in Figure 2, Bw-Tree consists of a mapping table that translates logical page identifiers to physical addresses, and a main tree structure with delta nodes that record incremental updates to pages. Delta records are periodically consolidated into CSR pages to prevent excessive read amplification. This design achieves lock-free concurrency control, enabling high-performance and multi-threaded access without contention. The delta chain approach amortizes random updates by appending incremental records, similar to log-structured writes in LSM-Trees [42], but without the overhead of global compaction. Note that periodic consolidation is still required to prevent excessive read amplification. While Bw-Tree has been adopted in some graph storage systems such as BG3 [58], these systems still organize vertices and neighbors based on IDs, leaving opportunities for topology-aware optimizations.

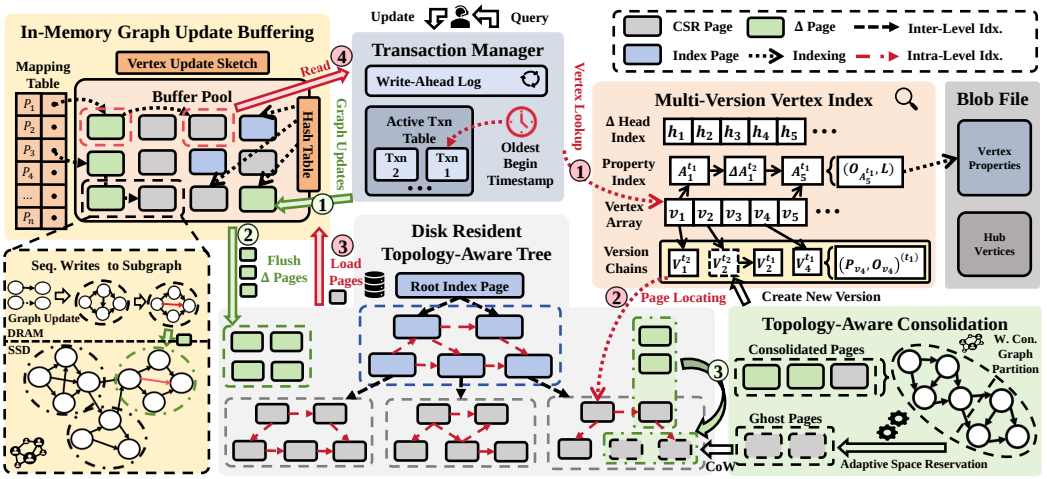


Fig. 3. The architecture of Bw-Graph. The example TAT contains three levels.

3 Bw-Graph Overview

This section presents the architecture of Bw-Graph, which is illustrated in Figure 3.

Architecture of Bw-Graph. Bw-Graph reconciles update efficiency and query performance in graph storage through four key components that collectively exploit the three optimization opportunities discussed above. ***Topology-Aware Tree (TAT)*** hierarchically co-locates topologically related vertices on disk to minimize I/O amplification during graph traversals and analytics. An in-memory ***Mapping Table*** bridges random graph updates and sequential disk I/O by employing Δ Page chains to transform scattered modifications into append-only operations. ***Multi-Version Vertex Index*** records each vertex's page ID, offset, and latest delta record location to enable single-hop navigation, while multi-versioning ensures that SMOs do not block concurrent read operations. The ***transaction manager*** maintains an active transaction table tracking ongoing transactions and a Write-Ahead Log (WAL) for recovery. Crucially, it guarantees *snapshot isolation*, ensuring the *atomic visibility* of multi-operation updates (e.g., cascading deletions) across interleaved workloads.

Write Operations (Phases ① – ③). Vertex and edge updates are first ① appended to the delta chain of the corresponding CSR page, which is maintained in the ***buffer pool***, before being asynchronously ② flushed to the disk-resident storage. When the length of the delta chain (θ_δ) exceeds a threshold (dynamically calibrated based on vertex access patterns), the system triggers ③ consolidation operations to merge delta records into the base CSR pages. If the consolidated page exceeds capacity, a topology-aware split divides the page while minimizing inter-page edges. Fundamentally, this hybrid structure of CSR pages and delta chains with consolidation transforms scattered graph updates into sequential writes to subgraphs, thereby optimizing the I/O access pattern.

Read Operations (Phases ① – ④). Neighbor scan is the dominant read operation in graph workloads [39, 57], directly determining overall query efficiency. To ensure snapshot isolation with minimal I/O overhead, Bw-Graph follows a version-aware retrieval path. First, the system ① queries the Multi-Version Vertex Index to ② resolve the target vertex version visible to the current transaction (based on the global start timestamp). This version entry points to the physical location of the base CSR page. The system accesses the buffer pool to retrieve this page. If the page is not cached, the system ③ fetches it from the disk-resident TAT. Simultaneously, the system looks up the in-memory Mapping Table to identify the corresponding delta chain that buffers recent updates for this CSR page. Finally, Bw-Graph ④ retrieves the complete neighbors by combining

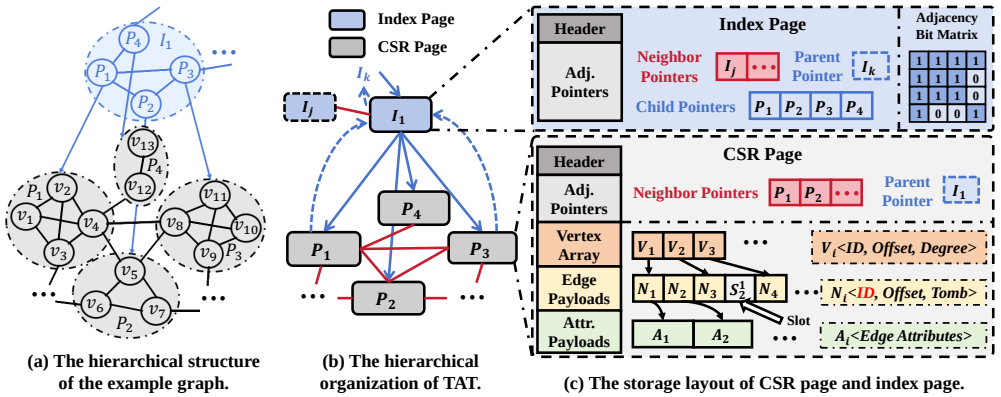


Fig. 4. The storage layout of TAT. (a) Multi-level hierarchical organization of an example graph. (b) TAT hierarchy overview. (c) Storage layouts for CSR and index pages (red IDs denote PTVs).

the edges from the base CSR page with the edge updates in the delta chain. Ultimately, this Paged CSR design minimizes read amplification to ensure low scan latency, while the topology-aware organization maximizes spatial locality, facilitating effective prefetching and high cache hit rates for graph analytics.

Property Graph Support. Beyond graph topology, Bw-Graph is designed to natively manage the property graph model. As illustrated in Figure 1(a), in addition to vertices and edges, Bw-Graph supports *vertex properties* (e.g., the *name* "Alien" and *year* "1979" for movie vertex v_5) and *edge properties* (e.g., the rating score 3 on the edge connecting v_2 to v_5). Specifically, vertex properties are managed in blob files, where each vertex entry in the Multi-Version Vertex Index maintains a direct pointer to its own property record (identified by offset and length) for fast random access. These properties are further chained via linked lists to support sequential scans, with updates appended as deltas to the blob files to avoid rewriting entire property records. Additionally, edge properties are maintained within CSR pages.

4 Topology-Aware Tree

In this section, we present the Topology-Aware Tree, a novel disk-resident tree structure that organizes graph data based on topological proximity rather than vertex IDs. We first describe the layout of the fundamental storage components of the TAT, including the ***CSR page*** and ***index page***. We then detail the hierarchical organization of the TAT and how graph data is distributed across multiple levels to exploit locality. Finally, we analyze the complexity of the TAT in terms of space consumption and construction time.

4.1 Storage Layout

TAT organizes graph data in a hierarchical structure where vertices and edges are partitioned based on their topological proximity to enhance spatial locality. A key design principle of TAT is the enforcement of a fixed page size for all nodes throughout the tree hierarchy, rather than using variable-sized buckets [54] for different communities. This fixed-size approach enables more predictable memory access patterns, simplified buffer management, and better cache utilization. Moreover, the fixed-size design ensures that TAT remains effective across diverse graph topologies regardless of community structure quality, making it particularly suitable for general-purpose graph storage systems. In this subsection, we describe the storage layout of TAT's fundamental components, including the CSR page and index page, as illustrated in Figure 4(a).

4.1.1 CSR page Layout. As depicted in Figure 4(c), each CSR page consists of a ***Header*** storing metadata (e.g., vertex/edge counts), an ***Adjacency Pointers*** array for topological navigation, and three ***data arrays*** for efficient storage. The ***Adjacency Pointers*** array facilitates tree navigation with *Neighbor Pointers* ($P_1, P_2, \dots$) for horizontal cross-page traversal and a *Parent Pointer* (I_k) for vertical upward navigation in the hierarchy. The three data arrays are organized in a ***slotted*** CSR format (where slots reserve space between vertex neighbors to accommodate edge insertions) as follows: (1) The ***Vertex Array*** stores fixed-length entries (V_i), each containing a vertex ID, degree, and an offset to its edge list; (2) The ***Edge Payload*** maintains neighbor entries (N_i) with target vertex IDs and *tombstone bits* (Tomb) for supporting logical deletion; (3) The ***Attribute Payload*** separates variable-length vertex and edge properties (A_i) from structural data, thereby optimizing cache locality during topology-only graph traversals. Figure 5 gives an example of a CSR page.

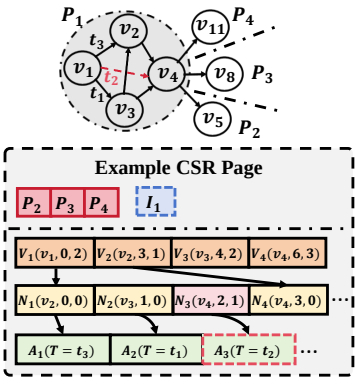


Fig. 5. A detailed example of a CSR page, illustrating how a subgraph and its edges are stored in continuous arrays.

4.1.2 Index Page Layout. As illustrated in the upper part of Figure 4(c), each index page maintains the same ***Header*** and ***Adjacency Pointers*** as CSR pages, but stores an ***Adjacency Bit Matrix*** instead of data arrays for efficient pruning during multi-hop traversal. The ***Adjacency Pointers*** array maintains *Child Pointers* ($P_1, P_2, \dots$) for hierarchical descent to sub-trees, alongside *Neighbor Pointers* (I_j) for horizontal cross-page traversal and *Parent Pointer* (I_k) for upward navigation. The ***Adjacency Bit Matrix*** encodes pairwise connectivity between child pages, where entry $M[i][j] = 1$ indicates that at least one vertex in page P_i has an edge to a vertex in page P_j. This compact bit-level representation enables $O(1)$ inter-page adjacency queries, facilitating rapid pruning of irrelevant search paths during multi-hop graph traversal.

4.1.3 Hub Vertex Management. Many real-world networks exhibit power-law degree distributions where a small fraction of *hub vertices* [28] possess extremely high degrees that exceed a single CSR page's capacity, requiring special handling. Inspired by value-separation strategies [32, 40], Bw-Graph stores hub vertex neighbor lists in a dedicated ***Blob File*** while CSR pages maintain only vertex metadata and offset pointers.

4.1.4 Degree-Stratified Compression. Compression reduces I/O volume but introduces decoding overhead, creating a space-latency trade-off in memory-constrained settings. Bw-Graph addresses this challenge via ***degree-stratified compression***, partitioning vertices into hub and regular tiers. For hub vertices in Blob Files, Bw-Graph applies ***differential-prefixed PForDelta*** [22] with gap-encoding and in-memory caching, as their large neighbor lists exhibit strong locality and are frequently accessed, making heavyweight compression both effective and amortized. For regular vertices in CSR pages, Bw-Graph employs ***Prefix-Tagged Varint (PTV)***, which embeds a 2-bit length indicator to directly encode integer width. Unlike traditional VByte [13], which requires sequential byte scanning via continuation bits and incurs branch mispredictions, PTV resolves integer width in a single step, eliminating such overhead. Meanwhile, reserved slots remain uncompressed with fixed-width encoding to enable direct offset-based access. This stratified approach efficiently handles diverse graph topologies while balancing competitive space efficiency and query performance.

Algorithm 1 Weight-Constrained Graph Partition

Input: Graph $G = (V, E)$, maximum partition weight $W_{\max}$
Output: Partition assignment $\mathcal{P}$

Phase 1: Graph Coarsening /* ε: predefined coarsening weight threshold */
$k \leftarrow \lceil \frac{\alpha_v \cdot |V| + \alpha_e \cdot |E|}{\varepsilon} \rceil$ /* α_v /α_e represent the weight of each vertex/edge */
$\mathcal{P}_{\text{coarse}} \leftarrow \text{KaMinPar}(G, k)$ /* Coarse-grained partitions */
Phase 2: Intra-Partition Graph Reordering
$\mathcal{O} \leftarrow \emptyset$ /* Ordered vertex sequences */
for all $P \in \mathcal{P}_{\text{coarse}}$ **do**
    $\mathcal{O}[P] \leftarrow \text{Gorder}(G[P])$ /* Locality-aware reordering */
end for
Phase 3: Sequential Vertex Packing
$\mathcal{P} \leftarrow \emptyset$ /* Fine-grained vertex groups */
for all $P \in \mathcal{P}_{\text{coarse}}$ **do**
    $g_{\text{curr}} \leftarrow \emptyset$, $w_{\text{curr}} \leftarrow 0$ /* Current group and weight */
    for each vertex $v \in \mathcal{O}[P]$ in order **do**
        $w_v \leftarrow \alpha_v + \alpha_e \cdot (|\mathcal{N}_G(v)| + s)$ /* s: reserved slots for future edge insertions */
        if $w_{\text{curr}} + w_v > W_{\max}$ **then**
            $\mathcal{P} \leftarrow \mathcal{P} \cup \{g_{\text{curr}}\}$, $g_{\text{curr}} \leftarrow \{v\}$, $w_{\text{curr}} \leftarrow w_v$ /* Start new group */
        else
            $g_{\text{curr}} \leftarrow g_{\text{curr}} \cup \{v\}$, $w_{\text{curr}} \leftarrow w_{\text{curr}} + w_v$
        end if
    end for
    if $g_{\text{curr}} \neq \emptyset$ **then**
        $\mathcal{P} \leftarrow \mathcal{P} \cup \{g_{\text{curr}}\}$ /* Add last group */
    end if
end for
return $\mathcal{P}$ /* Return fine-grained partition assignment */

4.2 Hierarchical Organization

TAT organizes graph data in a multi-level hierarchical structure to preserve topological locality, thereby enabling efficient subgraph-level navigation (e.g., graph sampling and ***subgraph prefetching***). As illustrated in Figures 4(a) and (b), the structure consists of two key components: (1) the bottom level comprises CSR pages, each storing a subset of vertices along with their adjacency lists in CSR format, and (2) the upper levels contain index pages that maintain the child pages and their topological adjacency relationships. This organization enables both spatial locality within CSR pages and efficient subgraph-level navigation through the index hierarchy. However, constructing such a hierarchy requires careful consideration of vertex assignment, neighbor page selection, and index construction to achieve both locality and partition balance.

Constructing TAT requires partitioning the graph into regions that preserve topological locality, and ensure each partition fits within a page size constraint. Two natural approaches exist, but face critical limitations. First, directly applying community detection or graph partitioning algorithms (e.g., METIS [38], KaMinPar [11], or Louvain [1]) produces high-quality partitions but lacks precise size control. Specifically, oversized partitions require recursive re-partitioning, creating numerous small fragments. When packed into fixed-size pages, these fragments introduce significant internal fragmentation and storage overhead. Second, performing graph reordering followed by sequential packing (e.g., Gorder [55]) guarantees size constraints but suffers from quadratic time complexity in the worst case, and yields low-quality partitions due to its greedy local optimization, which cannot guarantee globally coherent partitions. Neither approach achieves both high-quality partitioning and strict size guarantees efficiently. To address this challenge, Bw-Graph proposes a novel weight-constrained graph partitioning algorithm with both locality preservation and strict size guarantees.

Algorithm 2 Graph Sparsify

Input: Page adjacency graph $\mathcal{G} = (\mathcal{V}_\mathcal{G}, \mathcal{E}_\mathcal{G})$, max neighbor count α
Output: Sparsified graph $\mathcal{G}'$

Initialize $\mathcal{G}' \leftarrow (\mathcal{V}_\mathcal{G}, \emptyset)$, degree $d[p] \leftarrow 0\ \forall p$, Union-Find $\mathcal{U}$
$\mathcal{E}_{\text{sorted}} \leftarrow$ Sort edges in $\mathcal{E}_\mathcal{G}$ by $|\mathcal{N}_\mathcal{G}(u)| + |\mathcal{N}_\mathcal{G}(v)|$ descending /* Sort by endpoint degree sum */
Phase 1: Spanning Tree Construction
for each $(u, v) \in \mathcal{E}_{\text{sorted}}$ **do**
    if $\mathcal{U}$.Find$(u) \neq \mathcal{U}$.Find(v) **and** $d[u] < \alpha$ **and** $d[v] < \alpha$ **then**
        $\mathcal{G}' \leftarrow \mathcal{G}' \cup \{(u, v), (v, u)\}$ /* Add edge to sparsified graph */
        $d[u] \leftarrow d[u] + 1$; $d[v] \leftarrow d[v] + 1$
        $\mathcal{U}$.Union(u, v)
    end if
end for
Phase 2: Greedy Edge Filling
for each $(u, v) \in \mathcal{E}_{\text{sorted}}$ **do**
    if $(u, v) \notin \mathcal{E}_{\mathcal{G}'}$ **and** $d[u] < \alpha$ **and** $d[v] < \alpha$ **then**
        $\mathcal{G}' \leftarrow \mathcal{G}' \cup \{(u, v), (v, u)\}$ /* Fill remaining capacity */
        $d[u] \leftarrow d[u] + 1$; $d[v] \leftarrow d[v] + 1$
    end if
end for
return $\mathcal{G}'$ /* Return sparsified page adjacency graph */

4.2.1 Bottom Level Organization. At the bottom level, vertices are partitioned into disjoint sets and stored in CSR pages based on their topological proximity. To ensure balanced storage utilization and efficient buffer pool management, TAT enforces a ***vertex weight constraint*** during partitioning, where the sum of vertex weights within each partition must not exceed a predefined threshold $W_{\max}$. To achieve this balanced and high-quality partitioning, we propose a weight-constrained graph partitioning algorithm (detailed in Algorithm 1). First, the graph is coarsened using KaMinPar [11] to produce initial locality-aware partitions. Second, each partition is independently reordered using an optimized variant of Gorder [55] that replaces exhaustive candidate evaluation with an incremental max-heap approach, where vertex scores are updated incrementally as the sliding window advances, reducing per-iteration complexity from $O(|V_P| \cdot d_{\max})$ to $O(d_{\max} + \log |V_P|)$. Finally, vertices within each reordered partition are sequentially packed into groups, where a new group is created whenever adding a vertex would exceed $W_{\max}$. This algorithm preserves topological locality through KaMinPar-based coarsening and graph reordering, while guaranteeing strict page size constraints through sequential weight-aware packing. Each CSR page maintains a subset of vertices (e.g., $\{v_{12}, v_{13}\}$ or $\{v_1, v_2, v_3, v_4\}$ in Figures 4(a) and (b)) along with their outgoing edges.

4.2.2 Robustness Under Poor Community Structure. Our weight-constrained graph partitioning relies on a ***local*** reordering objective within each coarse partition, making the locality quality structurally independent of global community structure. In Phase 2 of Algorithm 1, Gorder reorders vertices within each coarse block C_j by greedily selecting v_{i+1} to maximize $|\mathcal{N}_{G[C_j]}(v_{i+1}) \cap \{v_1, \cdots, v_i\}|$, an operation that depends solely on the induced subgraph topology, rather than global modularity. Even if global modularity is low, real-world graphs consistently exhibit high local clustering coefficients, meaning dense micro-structures persist at the neighborhood level independent of macro-community quality. Gorder precisely exploits these micro-structures directly, ensuring high intra-partition edge density within each fine-grained partition regardless of how well-separated the coarse blocks are.

4.2.3 Neighbor Page Selection. Considering the limited capacity of the neighbor page array, not all adjacent pages can be recorded due to the fixed-size page layout. Therefore, neighbor pages must

Algorithm 3 Parallel TAT Construction

Input: Graph $G = (V, E)$, weight threshold $W_{\max}$, fanout $F_{\max}$, sparsification parameter α
Output: Hierarchical TAT structure $\mathcal{T}$

Phase 1: Bottom-Level CSR Page Construction
$\mathcal{P}_0 \leftarrow$ WeightConstrainedPartition$(G, W_{\max})$ /* Partition vertices by weight */
$C_0 \leftarrow \emptyset$ /* CSR page set at level 0 */
for each partition $P_i \in \mathcal{P}_0$ **do**
  $c_i \leftarrow$ CreateCSRPage(P_i)
  $C_0 \leftarrow C_0 \cup \{c_i\}$
end for
Phase 2: Page Adjacency Graph Construction
$\mathcal{G}_0 \leftarrow$ BuildPageAdjGraph(C_0, E) /* Build page adjacency graph */
$\mathcal{G}'_0 \leftarrow$ GraphSparsify$(\mathcal{G}_0, \alpha)$ /* Sparsify for prefetching */
for each CSR page $c_i \in C_0$ **do**
  $c_i.neighbors \leftarrow \mathcal{N}_{\mathcal{G}'_0}(c_i)$ /* Store neighbor pages */
end for
Phase 3: Hierarchical Index Construction
$\ell \leftarrow 0$, $C_\ell \leftarrow C_0$, $\mathcal{G}_\ell \leftarrow \mathcal{G}_0$
while $|C_\ell| > F_{\max}$ **do**
  $\ell \leftarrow \ell + 1$
  $\{C_{\ell,1}, \ldots, C_{\ell,k}\} \leftarrow$ WeightConstrainedPartition$(\mathcal{G}_{\ell-1}, W_{\max})$
  $\mathcal{I}_\ell \leftarrow \emptyset$ /* Index page set at level ℓ */
  for $j = 1$ to k **do**
    $\mathcal{G}_{\ell,j} \leftarrow$ InducedSubgraph$(\mathcal{G}_{\ell-1}, C_{\ell,j})$
    $i_{\ell,j} \leftarrow$ CreateIndexPage$(C_{\ell,j}, \mathcal{G}_{\ell,j})$
    for each page $p \in C_{\ell,j}$ **do**
      $p.parent \leftarrow i_{\ell,j}$ /* Link child to parent */
    end for
    $\mathcal{I}_\ell \leftarrow \mathcal{I}_\ell \cup \{i_{\ell,j}\}$
  end for
  $\mathcal{G}_\ell \leftarrow$ BuildPageAdjGraph$(\mathcal{I}_\ell, \mathcal{G}_{\ell-1})$ /* Next-level adjacency */
  $\mathcal{G}'_\ell \leftarrow$ GraphSparsify$(\mathcal{G}_\ell, \alpha)$
  for each index page $i_{\ell,j} \in \mathcal{I}_\ell$ **do**
    $i_{\ell,j}.neighbors \leftarrow \mathcal{N}_{\mathcal{G}'_\ell}(i_{\ell,j})$ /* Store neighbor pages */
  end for
  $C_\ell \leftarrow \mathcal{I}_\ell$ /* Treat index pages as next-level nodes */
end while
Phase 4: Root Index Page Creation
$root \leftarrow$ CreateIndexPage$(C_\ell, \mathcal{G}_\ell)$
for each page $p \in C_\ell$ **do**
  $p.parent \leftarrow root$
end for
return $\mathcal{T} = \{root, \{\mathcal{I}_\ell\}_{\ell=1}^{L-1}, C_0\}$ /* Return TAT hierarchy, L: total levels */

be carefully selected to maximize prefetching effectiveness while respecting the array capacity constraint. To address this challenge, Bw-Graph constructs a ***page adjacency graph***, where each node represents a CSR page (or index page) and each edge denotes cross-page connectivity induced by the original (lower-level) graph structure. A subset of edges is then greedily selected (detailed in Algorithm 2) to achieve high page coverage with minimal edge count. Specifically, the greedy algorithm sorts edges by the sum of endpoint degrees in descending order, and applies a two-pass selection strategy. The first pass ensures all CSR pages under the same index page form a connected component via union-find, guaranteeing full reachability through neighbor navigation. The second pass greedily fills remaining capacity to favor high-degree pages. Through these two passes, the

selected edges are recorded in the neighbor page array, enabling targeted prefetching of the most frequently co-accessed pages during graph traversal.

4.2.4 Upper Level Construction. Above the bottom level, index pages form a tree structure that captures the hierarchical partitioning of the graph through a bottom-up construction process (detailed in Algorithm 3). The construction proceeds in four phases. First, the graph is partitioned into CSR pages using the weight-constrained partitioning algorithm detailed in Algorithm 1 (Phase 1). Second, a page adjacency graph is constructed from cross-page edges and sparsified using Algorithm 2 (Phase 2). Third, pages are recursively partitioned based on this adjacency graph and grouped into index pages until the number of pages at the current level falls below the fanout threshold $F_{\max}$ (Phase 3). Finally, a root index page is created to cover all top-level pages (Phase 4). Each index page manages a set of child pages (either CSR pages at the lowest index level or other index pages at higher levels) and maintains two critical pieces of information: (1) the identifiers of all child pages in the page array, and (2) a neighbor page array storing adjacent pages identified through sparsification. This design enables efficient prefetching during graph traversal. When exploring neighbors of vertices in page P_i, the neighbor array directly reveals which other pages contain adjacent vertices, reducing unnecessary page accesses.

4.3 Complexity Analysis

In this subsection, the space complexity and construction time complexity of the TAT are analyzed.

Space Complexity. At Level 0, vertices are partitioned into $n_0 = O\left(\frac{|V|+|E|}{W_{\max}}\right)$ CSR pages, consuming $O(|V| + |E|)$ space. At each upper level $\ell \geq 1$, index pages manage pages from level $\ell - 1$ with fanout $\bar{F} = O(W_{\max})$, yielding $n_\ell = \lceil n_{\ell-1}/\bar{F} \rceil$. The hierarchy terminates in $L = O(\log_{\bar{F}} n_0)$ levels. Total index space is $\sum_{\ell=1}^{L} n_\ell \cdot S_{\text{index}} = O(n_0 \cdot S_{\text{index}})$, where S_{index} denotes the size of an index page. Since $S_{\text{index}} = O(F_{\max}^2) = O(\bar{F}^2)$, the index overhead is $O(n_0 \cdot F_{\max}^2) = O\left(\frac{(|V|+|E|)\cdot F_{\max}^2}{W_{\max}}\right) = O\left((|V| + |E|) \cdot F_{\max}\right)$.

Construction Time Complexity. Phase 1-2 requires $O(|V| + |E|)$ time: graph partitioning runs in $O(|V| + |E|)$ following KaMinPar [11], and both CSR page creation and page adjacency graph construction are linear to $|V| + |E|$. Phase 3 iterates for $L = O(\log_{\bar{F}} n_0)$ levels, with level ℓ requiring $O(n_\ell + m_\ell)$ time, where m_ℓ denotes edge count in $\mathcal{G}_\ell$. Since $n_\ell = O(n_{\ell-1}/\bar{F})$, the total vertex processing time is $\sum_{\ell=1}^{L} O(n_\ell) = O(n_0)$. After sparsification, $m_\ell = O(n_\ell \cdot \alpha)$, so the total edge processing time is $\sum_{\ell=1}^{L} O(n_\ell \cdot \alpha) = O(n_0 \cdot \alpha)$. Note that α is a system parameter bounded by a small constant in practice. In the worst case, the time complexity becomes $O(|V| + |E| + n_0 \cdot \alpha)$.

5 SMO in Bw-Graph

This section presents the ***SMOs*** in Bw-Graph. We first describe the storage layout of ***Delta Pages (Δ Pages)***. We then propose the ***topology-aware consolidation*** mechanism, along with the page splitting and merging operations it triggers.

5.1 Delta Pages

This subsection presents the storage layout of Δ Pages and how they manage graph updates in Bw-Graph.

5.1.1 Δ Page Storage Layout. As illustrated in Figure 6, each Δ Page consists of a header and an array of ***delta records***. The header stores metadata such as the current number of delta records. Delta records are organized as a contiguous array, where each record represents a single graph update operation. Each delta record maintains four components, including the operation (insertion,

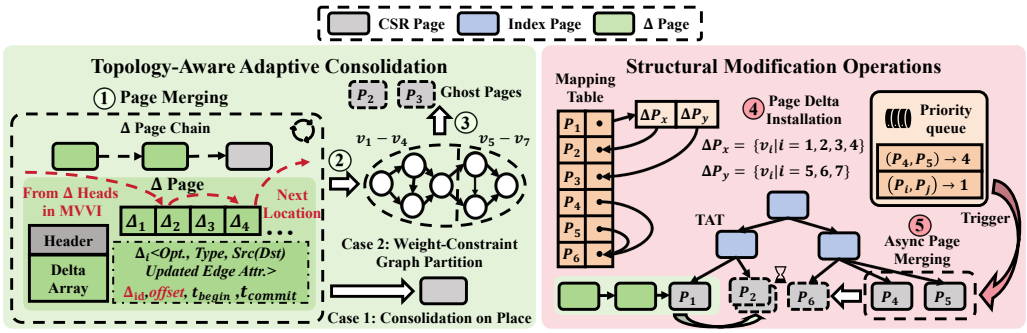


Fig. 6. The SMO in Bw-Graph. The delta records for each vertex are organized as a linked list. MVVI represents the Multi-Version Vertex Index.

modification, or deletion), the target type (vertex or edge), the identifiers of affected elements, and the updated edge attributes (for edges only). Vertex operations store the vertex ID, while edge operations store both source and destination vertex IDs (*src* and *dst*). For example, inserting an edge between v_1 and v_2 is represented as $\langle +, edge, v_1(v_2), \_ \rangle$. Additionally, each delta record attaches two timestamps, including the begin timestamp and the commit timestamp, both of which are used for MVCC-based transaction visibility management. To efficiently locate all delta records for a specific vertex without scanning the entire Δ Page, each record stores the location of the next record of the same vertex as a ⟨page ID, offset⟩ pair. The ***Multi-Version Vertex Index*** (detailed in § 6) stores the location of the most recent delta record. This design decouples logical graph updates from physical storage reorganization. Instead of triggering immediate structural changes, the Δ Page chain buffers updates logically, thereby avoiding frequent page splitting and amortizing the cost of TAT reconstruction across batched updates.

5.1.2 Adaptive Triggering. The Δ Pages form chains, representing sequential updates to a CSR page. As chains grow, read amplification from traversing multiple Δ Pages degrades query performance. Different from BG3 [58] which uses a fixed threshold θ_δ to trigger consolidation, Bw-Graph ***adaptively adjusts*** θ_δ based on workload patterns, i.e., vertices whose update frequency exceeds a threshold proportion are considered write-heavy and need to accumulate longer chains (large θ_δ) to amortize consolidation costs, while read-intensive vertices trigger early consolidation (small θ_δ of 1-2) to minimize read latency.

5.2 Topology-Aware Consolidation

To address read amplification from long delta chains while maintaining topology-aware organization during updates, Bw-Graph proposes a ***topology-aware consolidation mechanism*** that merges Δ Pages with their corresponding CSR pages and reorganizes pages to preserve graph locality. As illustrated in Figure 6, the consolidation process consists of five phases that ensure efficient structural modifications with minimal disruption to concurrent operations.

5.2.1 Δ Page Chain Consolidation. The consolidation process begins with ***Phase 1 (Page Merging)***, which extracts updates from the Δ Page chain and merges them with the base CSR page to produce an updated subgraph. Bw-Graph then evaluates whether the consolidated data fits within the original page capacity. If sufficient space remains, an in-place consolidation is performed without page splitting (Case 1 in Figure 6). Otherwise, when the consolidated data exceeds capacity, ***Phase 2 (Graph Partitioning)*** and ***Phase 3 (Ghost Page Packing)*** are triggered (Case 2). Unlike naive splitting by vertex ID, Phase 2 employs Algorithm 1 to ensure each resulting CSR page stores a

densely connected subgraph. In Phase 3, the partitioned data is packed into ghost pages, which are subsequently installed into the page table.

5.2.2 Latch-Free Structural Modifications. Inspired by Bw-Tree, Bw-Graph employs a ***latch-free approach*** to structural modifications. During page splitting, newly created pages are inserted as ***ghost pages***, visible to writers but inaccessible to readers (detailed below), allowing concurrent updates without blocking. As shown in Figure 6, when a CSR page splits into two partitions, a ***Page Delta*** (e.g., ΔP_x and ΔP_y) records the partitioning metadata (e.g., routing $v_1 - v_4$ to the new page P_2) and is atomically installed ***(Phase 4)*** into the Mapping Table via CAS. For page merging ***(Phase 5)***, Bw-Graph uses an asynchronous background thread that maintains a priority queue tracking update frequencies for page pairs. Pages are merged when cross-page edges between them occur frequently, and the combined data fits within capacity constraints, ensuring the merged page preserves graph locality. For instance, merging P_2 and P_3 redirects subsequent updates through the updated mapping without disrupting ongoing operations.

5.2.3 Workload-Driven Space Reservation. Since SMOs like page splitting are resource-intensive, minimizing their frequency is critical for sustained performance. However, traditional static reservation strategies (e.g., fixed 20% padding) are agnostic to workload patterns, inevitably leading to either insufficient space for write-intensive "hot" pages (causing frequent splits) or storage wastage on read-mostly "cold" pages. To address this limitation, Bw-Graph employs a ***gradient-based reservation heuristic*** during the consolidation. Specifically, based on the historical update velocity, reserved capacity is scaled proportionally to write intensity, ranging from zero overhead for stable pages to additional free slots for rapidly evolving ones. This workload-aware approach effectively balances the trade-off between write throughput and space utilization, ensuring that high-contention pages can absorb future deltas without triggering premature SMOs.

6 Version Control Protocol

This section proposes the version control mechanisms in Bw-Graph. We begin by presenting the ***Multi-Version Vertex Index***, which ensures SMOs remain transparent to concurrent readers. We then describe how Bw-Graph implements the MVCC protocol for transaction visibility management, leveraging the native versioning capability of Δ Pages.

6.1 Multi-Version Vertex Index

In Bw-Graph, the neighbor scans locate vertices through the Multi-Version Vertex Index rather than directly querying the mapping table. Consider a scenario where a consolidation thread splits CSR page c_i containing vertex v into two pages c_x and c_y. If a concurrent scan thread reads the Multi-Version Vertex Index to locate v, it may obtain a stale pointer to the original page c_i. Dereferencing this stale pointer accesses incorrect or partially updated data, causing erroneous results or crashes. A straightforward approach to maintaining consistency during SMO operations is to lock the affected pages and vertex index entries to prevent concurrent data access during updates, similar to the approaches employed in B^+-Trees and LSMGraph. However, this locking approach causes mutual blocking between readers and writers, as the held locks block concurrent read operations, significantly degrading throughput under high concurrency workloads. Inspired by Bw-Tree [25], Bw-Graph adopts a ***Multi-Version Vertex Index*** that eliminates blocking through lock-free design.

The multi-version design leverages Bw-Graph's non-in-place update strategy, where modifications append delta pages and structural changes create shadow pages asynchronously. In Bw-Graph, version control focuses exclusively on consolidation operations, since only consolidation-triggered page splits and merges relocate vertices, whereas updates are otherwise appended to Δ Pages without changing vertex locations. As illustrated in Figure 7, each vertex v_i maintains a `cur` (current)

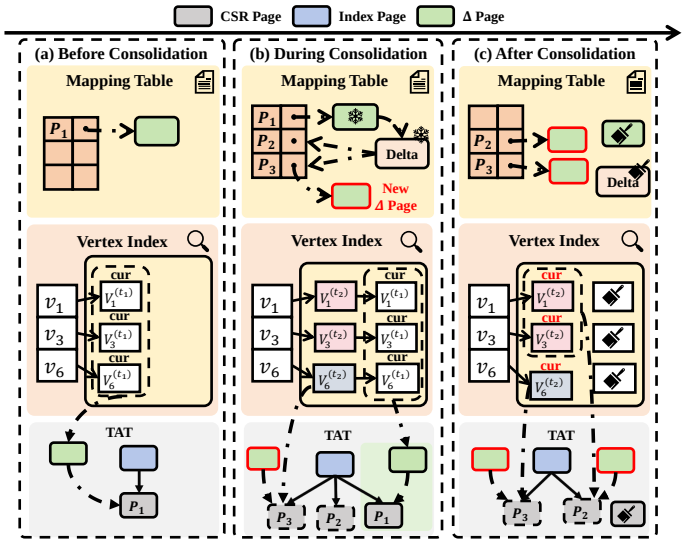


Fig. 7. Version control mechanism during page splitting. This example illustrates state transitions across the mapping table, vertex index, and TAT (a) before, (b) during, and (c) after consolidation.

pointer to its stable version (e.g., version $V_1^{(t_1)}$ before consolidation) containing the CSR page ID and the offset of the vertex. When consolidation relocates vertices to new pages (e.g., due to slot exhaustion or topology-aware splitting), new version entries (e.g., $V_1^{(t_2)}$ during consolidation) are created and linked to form version chains. After ghost pages are asynchronously populated, a CAS operation atomically advances each vertex's `cur` pointer to the new version, allowing concurrent readers to access stable versions without blocking the consolidation. Notably, the consolidations that update in-place without relocation require no version creation.

Figure 7 illustrates the version control protocol during consolidation. Initially, vertices v_1, v_3, and v_6 have their `cur` pointers referencing stable versions in CSR page P_1. When consolidation triggers a page split, new versions are created for affected vertices (e.g., $V_1^{(t_2)}$, $V_3^{(t_2)}$, $V_6^{(t_2)}$) and linked to their previous versions, forming version chains (e.g., $V_1^{(t_2)} \rightarrow V_1^{(t_1)}$). Concurrent scans follow the `cur` pointers to access stable versions (e.g., $V_1^{(t_1)}$), reading from the original page P_1 without blocking. Meanwhile, ghost pages are populated asynchronously, and the Mapping Table is updated to record the locations of the newly created pages. After the consolidation completes, CAS operations atomically advance the `cur` pointers to the new versions, marking the old page P_1 as obsolete for subsequent garbage collection. This protocol ensures that readers always access consistent data while structural modifications proceed concurrently without blocking.

6.2 MVCC in Bw-Graph

This subsection elaborates on the MVCC mechanism employed by Bw-Graph. While the Multi-Version Vertex Index ensures structural consistency by resolving vertex location shifts during SMOs, the Δ Page plays a complementary role by explicitly managing the visibility of data versions generated by concurrent transactions.

Beyond sequential writes to subgraphs, the Δ Page architecture is intrinsic to the version management strategy of Bw-Graph. Write transactions instantiate new versions by appending delta records to the Δ Page chain anchored at the corresponding CSR page. To optimize recovery efficiency and maintain a compact storage layout, Bw-Graph enforces a ***stability-based consolidation policy***.

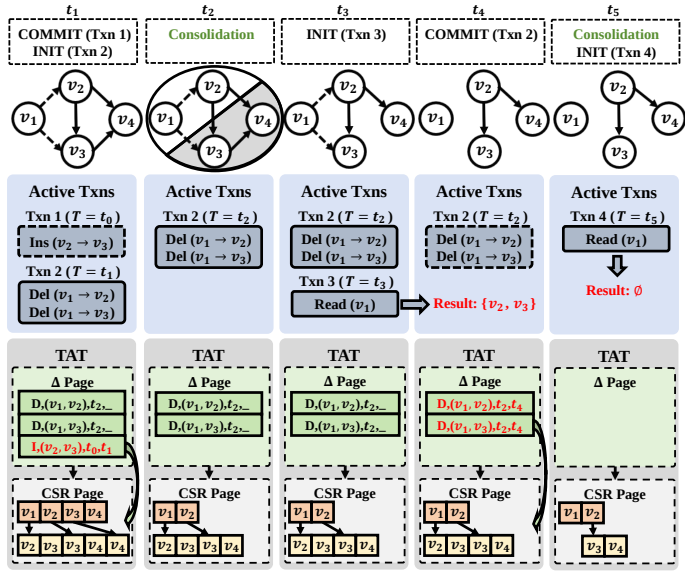


Fig. 8. A timeline illustrating concurrent transaction processing. It details the lifecycle of active transactions and their corresponding impact on the underlying TAT pages across distinct timestamps. $Read(v_1)$ represents reading the neighbors of vertex v_1. After t_2, vertices v_3 and v_4 are in an another CSR page, which is not shown in this figure. The dashed edges indicate deletions that have not yet been committed.

Specifically, the consolidation process selectively identifies stable versions, i.e., committed modifications with timestamps preceding the start time of the oldest active transaction. Only these stable records are materialized into the base CSR page. Since uncommitted versions remain exclusively in the Δ Page chain and are never materialized into CSR pages, rollback requires only discarding the corresponding delta records. This design obviates the need for complex undo logic on CSR pages.

Figure 8 exemplifies this lifecycle and the associated version consolidation strategy. The timeline traces the protocol's execution across five distinct timestamps. At t_1, Txn 1 commits an insertion which attains stability (highlighted in red), while Txn 2 is initialized. By t_2, a consolidation reorganizes the physical layout (grouping v_1 and v_2 into the depicted CSR page). Crucially, while the stable insertion from Txn 1 is materialized, the multi-operation updates from the active Txn 2 (specifically, two deletion records targeting v_1) are preserved in the Δ Page chain. When Txn 3 performs a read at t_3, it strictly adheres to ***snapshot isolation*** by traversing the delta chain and bypassing the uncommitted records from Txn 2 to retrieve the visible version $\{v_2, v_3\}$. Upon the commit of Txn 2 at t_4, its deletion records satisfy the stability criteria, ensuring that all deletion records from Txn 2 become visible atomically upon commit. Subsequently, the consolidation at t_5 materializes these deletions into the CSR page, allowing Txn 4 to correctly read the updated result.

7 Evaluation

This section presents the experimental evaluation of Bw-Graph. We first describe the experimental setup. We then evaluate Bw-Graph across four graph workloads, including transactional processing for update ingestion and neighbor scans, analytical processing for graph algorithms, hybrid workload combining concurrent updates with analytics, and graph sampling for graph learning support. Moreover, we assess the storage footprint of Bw-Graph. Finally, we conduct design validations to justify our design choices and demonstrate the effectiveness of our proposed techniques.

Table 2. Statistics of the used datasets. $|V|$, $|E|$ are the numbers of vertices and edges, respectively.

Dataset	$\|V\|$	$\|E\|$	Avg. Degree	Max. Degree	Size
Road Network (RN)	1 M	1.5 M	1.42	9	39 MB
YouTube (YT)	1.1 M	3 M	2.58	28,754	58 MB
LiveJournal (LJ)	4 M	34 M	8.59	14,815	600 MB
Orkut (OT)	3 M	117 M	38.14	33,313	1.9 GB
Datagen-8_8-ZF (DG)	168 M	413 M	2.46	2,242	10 GB
IT-2004 (IT)	41 M	1.02 B	24.88	1,326,744	20 GB
UK-2007 (UK)	105 M	3.73 B	31.18	975,419	70 GB

7.1 Experimental Setup

We describe the system configuration, datasets, and competitors in this subsection.

System Configuration. The experiments are carried out on a server with an Intel Xeon E5-2630 2.20GHz CPU, 256GB DRAM, 1.5TB disk, and Ubuntu 20.04 operating system. Bw-Graph is implemented in approximately 20,000 lines of C++20 code [1] and compiled with GCC 13.1 using O3 level optimization. For each dataset, the available memory of the buffer pool in Bw-Graph is allocated as half of the dataset size. For all experiments, tasks exceeding 3,600 seconds are terminated with their execution time recorded as 3,600 seconds.

Datasets. We use seven graph datasets spanning multiple domains in our experiments, as shown in Table 2. IT and UK are large-scale Web graphs representing hyperlink structures from the Laboratory for Web Algorithms [3]. RN is a road network dataset from the Stanford Large Network Dataset Collection [23]. DG is a synthetic graph generated for LDBC benchmarking [15] with controlled characteristics. LJ, OT, and YT are social networks from the Stanford Large Network Dataset Collection [23], capturing user interactions across different platforms.

Competitors. We compare Bw-Graph against seven representative graph systems:

- ***Aster*** [39]: A graph storage system built on LSM-Tree. For a fair comparison, we evaluate against its storage engine, Poly-LSM, directly.
- ***LiveGraph*** [61]: A transactional graph storage system based on adjacency lists and edge lists, designed to support concurrent updates and queries efficiently.
- ***LLAMA*** [34]: A CSR-based graph storage system that stacks CSR snapshots to enable fast graph updates while maintaining efficient read performance.
- ***Neo4j*** [41]: An industry-standard graph database system based on the linked list, which is widely adopted in production environments.
- ***Kuzu*** [20]: An embedded graph database system designed for high-performance graph analytics, utilizing columnar CSR-based adjacency lists to accelerate graph algorithms.
- ***GraphChi*** [21]: A pioneering single-machine out-of-core graph processing system that uses parallel sliding windows to partition and store graphs for accelerated computation.
- ***GridGraph*** [62]: A single-machine graph processing system based on grid partitioning for efficient graph analytics.

These systems reflect a diverse spectrum of design choices, including graph storage engines built upon LSM-tree, adjacency-list, and CSR-based structures, widely-adopted industrial graph database systems, and graph processing systems dedicated to large-scale graph analytics.

7.2 Transactional Processing

In this subsection, we evaluate the transactional processing performance of Bw-Graph, focusing on graph update throughput and neighbor scan latency.

[1] https://github.com/yaoyaowong/Bw-Graph

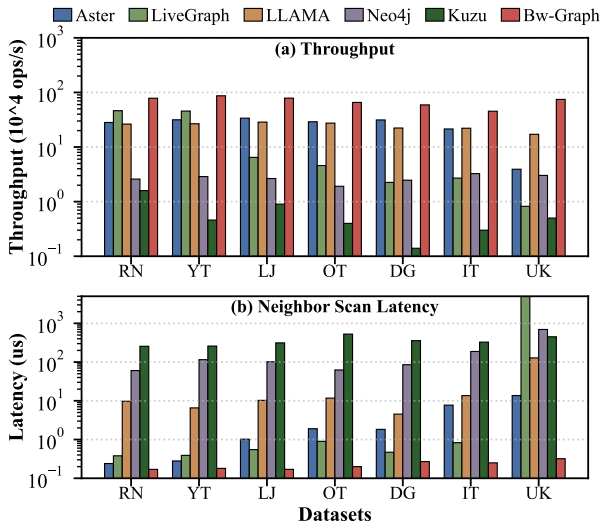


Fig. 9. The throughput and latency under transactional workloads on different datasets and graph storage systems.

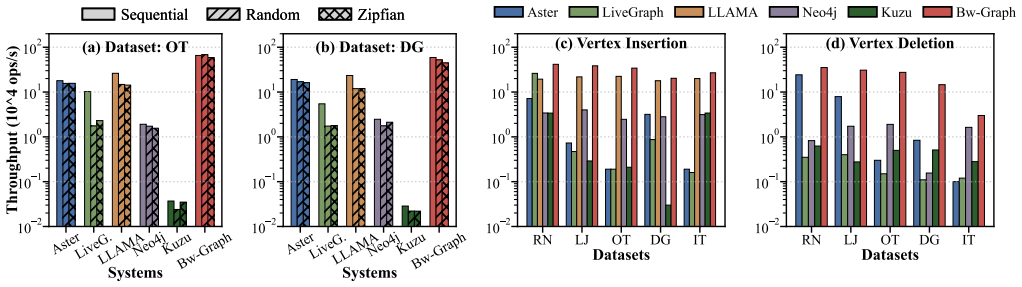


Fig. 10. The throughput under different edge insertion patterns.

Graph Update Throughput. We evaluate the transactional processing performance of inserting the remaining 20% and randomly deleting 10% of all edges, after first importing 80% of the edges as the base graph. For systems that do not support edge deletion (specifically, LLAMA [34]), only edge insertions are performed. Figure 9(a) presents the update throughput of the evaluated systems across different datasets. As shown, Bw-Graph consistently outperforms other graph storage systems across all datasets. For instance, on the UK dataset, Bw-Graph achieves a throughput of 745K ops/s, which is 19× higher than that of the LSM-based Aster (39.2K ops/s). This superior performance can be attributed to the Δ Pages attached to the corresponding CSR pages. Specifically, all vertex updates are appended to the Δ Page chain, which is periodically flushed to disk, converting random writes into sequential writes in a log-structured manner.

Neighbor Scan Latency. Neighbor scan is a fundamental operation that, given a vertex, retrieves all its neighbors through the outgoing edges. To evaluate the neighbor scan performance, we randomly select 10% of the vertices from the graph and scan their neighbors, measuring the average latency across all queries. As shown in Figure 9(b), Bw-Graph demonstrates low latency across most datasets. For example, on the DG dataset, Bw-Graph achieves the neighbor scan latency of 0.27 μs, a 16.8× reduction compared to LLAMA (4.53 μs). Notably, as the dataset scale increases, Bw-Graph

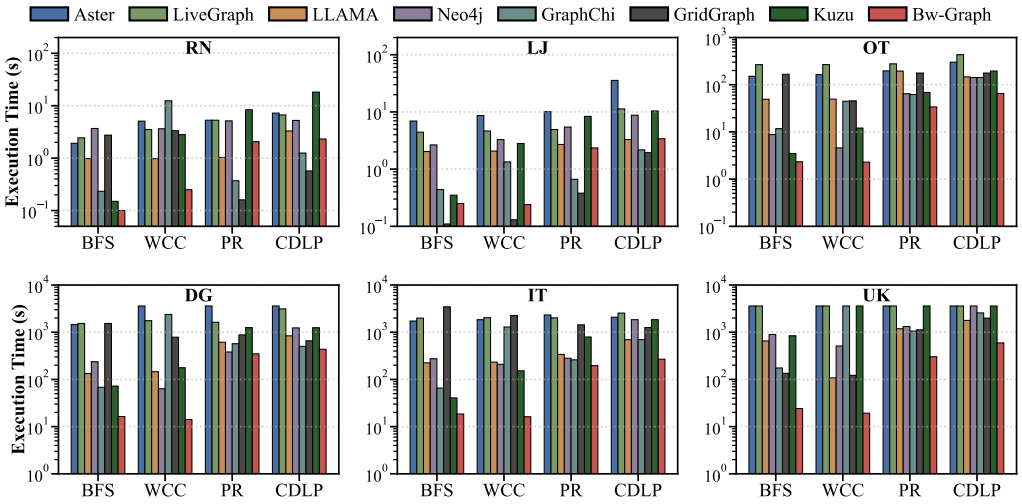


Fig. 11. The latency of graph algorithms on different datasets and graph storage systems.

maintains consistently low neighbor scan latency, which can be attributed to its Multi-Version Vertex Index that enables direct access to vertex neighbors.

Varying Edge Insertion Patterns. To evaluate Bw-Graph's resilience under different update patterns, we perform edge insertion on the baselines under three modes, including sequential (the inserted edges are sorted by vertex ID), random (the inserted edges are shuffled), and Zipfian (the inserted edges are concentrated on hotspot source vertices or subgraphs). Figures 10(a) and (b) present the results. Clearly, Bw-Graph maintains competitive throughput across all three patterns. Especially under the Zipfian workload, which could directly model page bombardment, Bw-Graph achieves a throughput of 587K ops/s, outperforming Aster by 3.8× (156K ops/s). This advantage stems from Bw-Graph's ability to coalesce scattered updates into sequential writes to subgraphs, naturally accommodating localized high-frequency modifications.

Vertex Operations. We evaluate the vertex operations by inserting a portion of new vertices (each connected to k randomly selected existing vertices, where k equals the graph's average degree) and deleting a portion of vertices, which triggers cascading edge removals. Figures 10(c) and (d) confirm Bw-Graph's superiority across the two vertex operations. For vertex insertions, Bw-Graph achieves a throughput of 203K ops/s on DG, outperforming Aester by 6.4× (31.8K ops/s). For vertex deletions, on the OT dataset, Bw-Graph achieves a throughput of 275K ops/s compared to LiveGraph's 1.1K ops/s, a 250× improvement. Notably, vertex deletion throughput decreases as dataset scale grows. This trend is attributable to larger datasets exhibiting higher average degrees, as higher degrees amplify cascading edge removals. The reason for the performance advantage is two-fold. First, Bw-Graph accelerates insertions (edge and vertex alike) by buffering random writes in Δ Pages, which are later flushed through sequential writes to amortize the overhead of random I/O. Second, for vertex deletions, Bw-Graph exploits the topological locality of cascading edge removals to coalesce scattered updates into sequential writes to subgraphs. This transformation avoids the random I/O overhead in ID-based systems.

7.3 Analytical Processing

We measure the execution time of four representative graph algorithms, specifically Breadth-First Search (BFS), Weakly Connected Components (WCC), PageRank (PR), and Community Detection

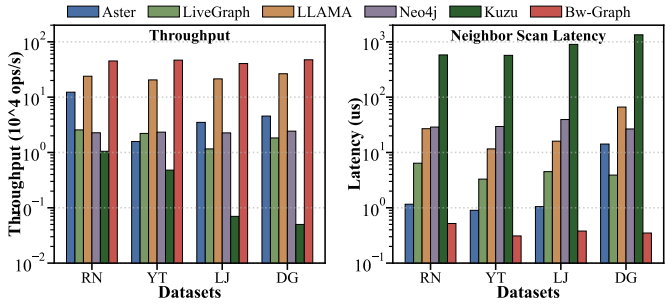


Fig. 12. The throughput and latency under hybrid workloads on different datasets and graph storage systems.

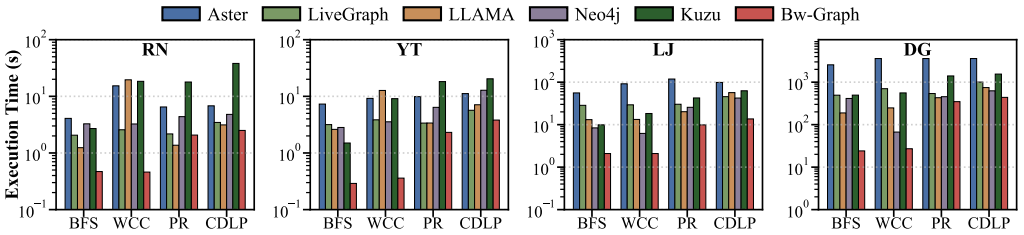


Fig. 13. The latency of different graph algorithms under hybrid workloads on different datasets and graph storage systems.

using Label Propagation (CDLP). These algorithms, covering diverse traversal patterns and computational characteristics, are widely adopted in graph analytics benchmarks, and thus provide a comprehensive assessment of read performance for graph storage systems.

Figure 11 presents the execution time of the four algorithms across different datasets. For instance, on the UK dataset, Bw-Graph completes the WCC in 19.3 s, which is 6.3× faster than GridGraph (122 s), a specialized graph processing system. This advantage stems from two key factors. First, the paged CSR structure accelerates edge scan operations by enabling efficient sequential access to neighbors within each page. Second, the topology-aware organization fully exploits spatial locality by clustering tightly connected vertices together, which maximizes cache hit rates during edge scanning. Furthermore, this design enables effective and accurate prefetching through asynchronous I/O, which overlaps computation with data loading.

7.4 Hybrid Workload

We evaluate the system performance under hybrid workloads, where read operations (neighbor scans and the four graph algorithms) and write operations are executed concurrently. Specifically, we measure the neighbor scan latency and the analytical algorithm execution time while the system is subjected to continuous write pressure.

Figure 12 presents the update throughput and the neighbor scan latency of evaluated systems under hybrid workloads across different datasets, and Figure 13 presents the execution time of the four graph algorithms under concurrent updates. The experimental results demonstrate that Bw-Graph exhibits outstanding capability in handling HTAP workloads. For example, on the DG dataset under concurrent neighbor scans, Bw-Graph achieves a throughput of 474K ops/s, which is 25.9× higher than LiveGraph (18.3K ops/s). Moreover, even under concurrent graph updates, Bw-Graph completes BFS on the DG dataset in 24.1 seconds, which is 106.3× faster than Aster

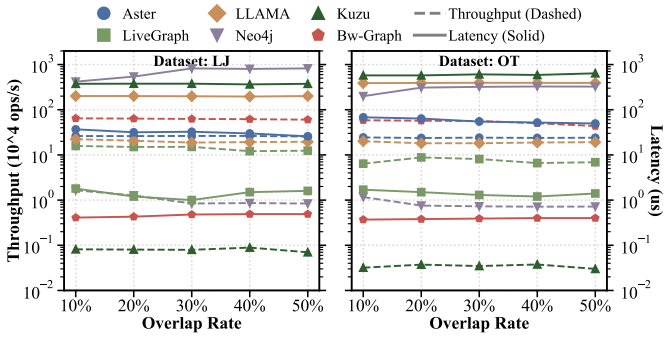


Fig. 14. The throughput and latency under hybrid workloads with different read-write overlap.

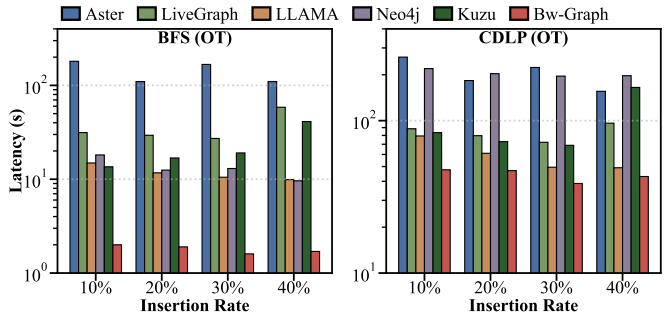


Fig. 15. The latency of different graph algorithms under hybrid workloads on different edge insertion rates.

(2561 s). This superior performance can be attributed to our latch-free SMOs and Multi-Version Vertex Index. Specifically, the latch-free SMO adapted from Bw-Tree mitigates synchronization bottlenecks in concurrent access to graph storage systems, and the Multi-Version Vertex Index reduces the blocking effects of SMOs on concurrent read operations.

Varying Read-Write Overlap Rate. To clearly understand the system performance under varying degrees of read-write conflicts, we control the overlap ratio (10%, 20%, ..., 50%) between queried vertices and their updated neighbors in hybrid workloads. Figure 14 shows the results. Bw-Graph maintains the competitive performance across all conflict levels. For instance, at 50% read-write overlap on the OT dataset, Bw-Graph achieves a neighbor scan latency of 0.4 μs, outperforming Aster by 124× (49.6 μs). This superior performance stems from Bw-Graph's Multi-Version Vertex Index, which enables non-blocking reads that avoid interference with SMOs.

Varying Insertion Ratios. To demonstrate the robustness of Bw-Graph across different workload configurations, we vary the insertion ratio from 10% to 40% in mixed workloads comprising concurrent edge insertions and graph algorithm executions. As shown in Figure 15, Bw-Graph maintains the competitive performance across all ratios. The results demonstrate that Bw-Graph can sustain high throughput across workloads with varying insertion ratios, where higher ratios introduce more intensive structural changes to the graph.

7.5 Storage Footprint

Storage efficiency is a critical factor in graph storage systems, as real-world graphs often contain billions of vertices and edges, leading to substantial storage overhead. To evaluate the storage

Table 3. Storage footprint comparison (disk space in MB/GB). **Bold** denotes the best result, underline denotes the second best.

Dataset	LLAMA	LiveGraph	Aster	Neo4j	Kuzu	GraphChi	GridGraph	Bw-Graph
RN	28.5 MB	334 MB	45 MB	189 MB	111 MB	42 MB	95 MB	**27 MB**
YT	40.5 MB	572 MB	69 MB	478 MB	157 MB	74 MB	142 MB	**36 MB**
LJ	326.3 MB	5.2 GB	622 MB	2.5 GB	1.2 GB	957 MB	2.1 GB	**311 MB**
OT	941 MB	17 GB	1.9 GB	5.8 GB	4.5 GB	3.3 GB	5.3 GB	**914 MB**
DG	**4.1 GB**	90 GB	5.9 GB	8.4 GB	24.62 GB	15 GB	19 GB	5.1 GB
IT	8.3 GB	139 GB	8.9 GB	19.6 GB	33.9 GB	34 GB	53 GB	**6.4 GB**
UK	26 GB	257 GB	29 GB	120 GB	105.6 GB	111 GB	151 GB	**19.2 GB**

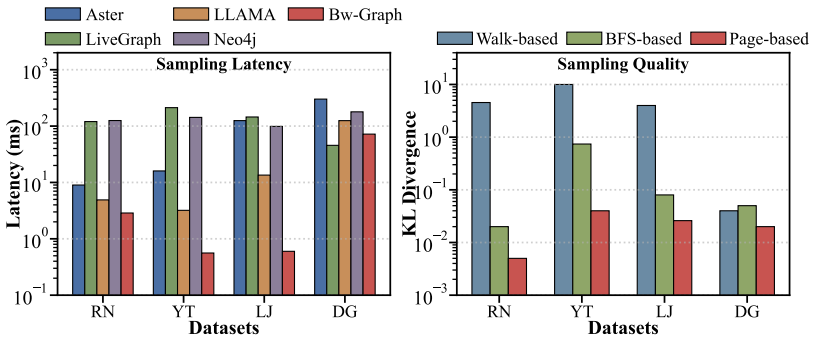


Fig. 16. The latency and quality of graph sampling on different datasets and graph storage systems.

footprint, we fully load each graph dataset into the respective graph storage systems and measure the total disk space consumed by the database storage files after the loading process completes.

Table 3 presents the storage footprint of different graph storage systems. Bw-Graph achieves competitive space efficiency across different graph datasets. For instance, on the YT dataset, Bw-Graph consumes a space of 36MB, approximately 90% of LLAMA's footprint (40MB). While on the RN dataset, Bw-Graph uses 27MB versus LLAMA's 28.5MB. This efficiency stems from our degree-stratified compression strategy, which partitions vertices into hub and regular tiers with differentiated compression strategies. This stratified approach efficiently stores diverse graph data on disk, from power-law social networks (e.g., YT and UK) to sparse road networks (e.g., RN).

7.6 Graph Sampling

Graph sampling is widely used in practice to approximate large-scale graph query results through partial graph traversal, avoiding the overhead of scanning the entire graph. In Bw-Graph, the TAT enables efficient graph sampling by loading adjacent pages based on graph topology, allowing the system to traverse and sample connected subgraphs naturally. We implement both random walk-based and BFS-based graph sampling algorithms and compare their performances against the sampling algorithm in Bw-Graph. Specifically, Bw-Graph employs a page-based sampling strategy that begins by identifying an index page, and then expands (via a BFS-like traversal over child pages) the index page layer by layer following the page topology until the required number of vertices is collected. The experimental results report the latency of the best-performing sampling method for each system. Beyond measuring the latency, we evaluate sampling quality by computing the

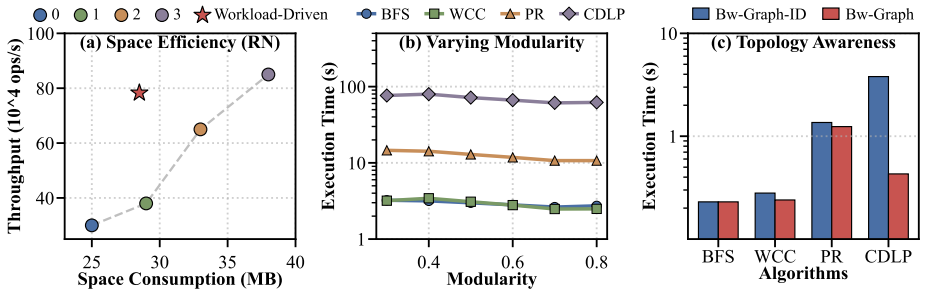


Fig. 17. The results of the design validation.

Kullback-Leibler (KL) divergence [18] between the degree distributions of the sampled subgraph and those of the original graph.

Figure 16 presents the sampling latency and quality of evaluated systems and algorithms across different datasets. The experimental results demonstrate that Bw-Graph obtains higher-quality subgraphs within less execution time compared to baseline systems. Note that Bw-Graph exhibits higher latency than LiveGraph on the DG dataset, as LiveGraph is an in-memory graph storage system that incurs no I/O overhead. The low KL divergence values indicate that the sampled subgraphs accurately preserve the degree distribution characteristics of the original graphs, which is crucial for maintaining the statistical properties needed by downstream applications. This efficient sampling capability is particularly valuable to out-of-core GNN training systems [26], where mini-batch training requires sampling subgraphs from the disk-resident graph data.

7.7 Design Validation

We validate three key design decisions, including adaptive space reservation, weight-constrained graph partitioning, and topology-aware organization.

Space Reservation Strategy. We compare our adaptive strategy against fixed reservation policies (0, 1, 2, and 3 slots per vertex) regarding space usage and throughput of the edge insertion on the RN dataset. The experimental results shown in Figure 17(a) demonstrate that our adaptive strategy achieves a superior trade-off, reducing space consumption by 29% compared to the fixed 3-slot policy (27 MB vs. 38 MB) while retaining 92% of the throughput (783K vs. 850K ops/s).

Weight-constrained Graph Partitioning. Since Bw-Graph's topology-aware organization is designed as a general-purpose graph storage method, it should remain effective across graphs with varying community structure quality. To this end, we adopt weight-constrained graph partitioning and assess its effectiveness by measuring Bw-Graph's sensitivity to community quality across varying modularity [1] levels. Specifically, we synthesize six graphs using FastSGG [52] with identical vertex and edge counts but controlled modularity levels ranging from 0.3 to 0.8, isolating the community quality as the sole variable. Figure 17(b) shows that Bw-Graph maintains consistent performance regardless of the modularity. For instance, the WCC runtime exhibits negligible fluctuation (3.19s at 0.3 vs. 3.09s at 0.5), with only a 3.2% variation. The experimental result confirms that our weight-constrained partitioning exploits local connectivity patterns, which are independent of global community structure, making Bw-Graph applicable to diverse real-world graphs.

Topology-Aware Organization. The topology-aware organization is compared against an ID-based organization (i.e., direct vertex packing without weight-constrained graph partitioning) to validate the benefits of our TAT. The experimental results in Figure 17(c) show that the TAT accelerates graph algorithms across all workloads on the YT dataset. For instance, the execution

time of CDLP reduces from 3.8s to 0.43s (8.8× speedup). This advantage stems from the TAT's ability to co-locate vertices with dense connections, reducing random I/O and improving cache locality during traversals.

8 Related Work

Graph storage systems form the foundation of graph processing, and have been explored across a wide design space of storage and computation paradigms. Distributed graph processing systems such as Pregel [35], PowerGraph [10], GraphX [56], GraphLab [31], GBase [17], Chaos [47], and Gemini [60] employ distributed storage and computation with graph partitioning to achieve scalability across clusters, but require substantial infrastructure, unsuitable for single-machine deployments. In-memory graph processing systems, including GraphOne [19], RisGraph [8], KickStarter [50], LiveGraph [61], Spruce [49], and RapidStore [12], exploit high-speed memory access for low-latency processing, but are constrained by memory capacity and volatility. To overcome these limitations, disk-resident graph processing systems such as GraphChi [21], XStream [48], GridGraph [62], PathGraph [51], DGraph [59], Mosaic [33], GraphSSD [37], ACGraph [4], and CAVE [43] process graphs using secondary storage, employing locality optimization to transform random accesses into sequential reads, parallelism with concurrency control, and pruning strategies to reduce I/O overhead. Beyond computation, the design of the storage layer itself presents distinct challenges, motivating a dedicated line of graph storage systems optimized for efficient data ingestion and access. Temporal graph systems like Clock-G [36] and ChronoGraph [7] focus on evolving graphs. Recently, LSM-Tree-based systems, including LSMGraph [57], Aster [39], and BACH [14], have emerged as a popular approach to disk-resident graph storage. Despite their write efficiency, these systems suffer from read amplification due to multi-level lookups across LSM-Tree levels. Alternative approaches include CSR-based systems like LLAMA [34], Teseo [6], and hybrid structures in LiveGraph [61]. Unlike prior work, Bw-Graph addresses both the read amplification inherent in LSM-tree-based systems and the scattered update overhead in ID-based-organized storage systems.

9 Conclusion

We propose Bw-Graph, a graph storage system that addresses fundamental limitations in existing approaches by harmonizing a TAT with paged CSR. By designing CSR pages for efficient neighbor scans and proposing Δ Pages for sequential subgraph writes, Bw-Graph achieves high write throughput and low read latency. The TAT employs a hierarchical organization that exploits graph locality while maintaining balanced structural organization across diverse graph topologies. Furthermore, a tailored MVCC mechanism eliminates blocking between structural maintenance and concurrent operations, enabling high-performance graph transactional and analytical processing without sacrificing consistency. The experimental evaluation demonstrates that Bw-Graph delivers significant performance improvements across diverse workloads. Future work includes extending Bw-Graph to support efficient GNN training by leveraging its topology-aware organization for optimizing graph sampling and feature aggregation.

Acknowledgments

The authors would like to thank the anonymous reviewers for their insightful comments and suggestions on this work. The authors also thank Furong Chen and the team at Tencent for their support. This work is supported in part by the National Natural Science Foundation of China (No. 62372264 and No. 92467203). Chaokun Wang (chaokun@tsinghua.edu.cn) serves as the corresponding author.

References

[1] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. 2008. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment* 2008, 10 (oct 2008), P10008. doi:10.1088/1742-5468/2008/10/P10008

[2] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. 2004. UbiCrawler: a scalable fully distributed web crawler. *Softw. Pract. Exper.* 34, 8 (July 2004), 711–726. doi:10.1002/spe.587

[3] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework I: Compression Techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*. ACM Press, Manhattan, USA, 595–601.

[4] Dechuang Chen, Sibo Wang, and Qintian Guo. 2025. ACGraph: An Efficient Asynchronous Out-of-Core Graph Processing Framework. *Proc. ACM Manag. Data* 3, 6, Article 290 (Dec. 2025), 26 pages. doi:10.1145/3769755

[5] Aaron Clauset, M. E. J. Newman, and Cristopher Moore. 2004. Finding community structure in very large networks. *Phys. Rev. E* 70 (Dec 2004), 066111. Issue 6. doi:10.1103/PhysRevE.70.066111

[6] Dean De Leo and Peter Boncz. 2021. Teseo and the analysis of structural dynamic graphs. *Proc. VLDB Endow.* 14, 6 (Feb. 2021), 1053–1066. doi:10.14778/3447689.3447708

[7] Benjamin Erb, Dominik Meißner, Jakob Pietron, and Frank Kargl. 2017. Chronograph: A Distributed Processing Platform for Online and Batch Computations on Event-sourced Graphs. In *Proceedings of the 11th ACM International Conference on Distributed and Event-Based Systems* (Barcelona, Spain) *(DEBS '17)*. ACM, NY, USA, 78–87. doi:10.1145/3093742.3093913

[8] 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 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. ACM, NY, USA, 513–527. doi:10.1145/3448016.3457263

[9] M. Girvan and M. E. J. Newman. 2002. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences* 99, 12 (2002), 7821–7826. arXiv:https://www.pnas.org/doi/pdf/10.1073/pnas.122653799 doi:10.1073/pnas.122653799

[10] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Hollywood, CA, USA) *(OSDI'12)*. USENIX Association, USA, 17–30.

[11] Lars Gottesbüren, Tobias Heuer, Peter Sanders, Christian Schulz, and Daniel Seemaier. 2021. Deep Multilevel Graph Partitioning. In *29th Annual European Symposium on Algorithms, ESA 2021 (LIPIcs, Vol. 204)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 48:1–48:17. doi:10.4230/LIPIcs.ESA.2021.48

[12] Chiyu Hao, Jixian Su, Shixuan Sun, Hao Zhang, Sen Gao, Jianwen Zhao, Chenyi Zhang, Jieru Zhao, Chen Chen, and Minyi Guo. 2025. RapidStore: An Efficient Dynamic Graph Storage System for Concurrent Queries. *Proc. VLDB Endow.* 18, 10 (Sept. 2025), 3587–3600. doi:10.14778/3748191.3748217

[13] William Hersh. 2001. Managing gigabytes–compressing and indexing documents and images. *Information Retrieval* 4, 1 (2001), 79.

[14] Jianfeng Huang, Yihao Cao, Shubing Ren, Baohua Wu, and Dongjing Miao. 2025. BACH: Bridging Adjacency List and CSR Format Using LSM-Trees for HGTAP Workloads. *Proc. VLDB Endow.* 18, 5 (Aug. 2025), 1509–1521. doi:10.14778/3718057.3718076

[15] Alexandru Iosup, Tim Hegeman, Wing Lung Ngai, Stijn Heldens, Arnau Prat-Pérez, Thomas Manhardto, Hassan Chafio, Mihai Capotă, Narayanan Sundaram, Michael Anderson, Ilie Gabriel Tănase, Yinglong Xia, Lifeng Nai, and Peter Boncz. 2016. LDBC graphalytics: a benchmark for large-scale graph analysis on parallel and distributed platforms. *Proc. VLDB Endow.* 9, 13 (Sept. 2016), 1317–1328. doi:10.14778/3007263.3007270

[16] it 2004. 2004. IT-2004. https://law.di.unimi.it/webdata/it-2004/. https://law.di.unimi.it/webdata/it-2004/ Italian Web Graph from IIT-CNR.

[17] 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* (San Diego, California, USA) *(KDD '11)*. ACM, NY, USA, 1091–1099. doi:10.1145/2020408.2020580

[18] Solomon Kullback and Richard A Leibler. 1951. On information and sufficiency. *The Annals of Mathematical Statistics* 22, 1 (1951), 79–86.

[19] 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 (jan 2020), 40 pages. doi:10.1145/3364180

[20] Kuzu. 2026. https://github.com/kuzudb/kuzu (last checked on 2026-01).

[21] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-Scale Graph Computation on Just a PC. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. USENIX Association, Hollywood, CA, 31–46. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/kyrola

[22] D. Lemire and L. Boytsov. 2015. Decoding billions of integers per second through vectorization. *Softw. Pract. Exper.* 45, 1 (Jan. 2015), 1–29. doi:10.1002/spe.2203

[23] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[24] Ian X. Y. Leung, Pan Hui, Pietro Liò, and Jon Crowcroft. 2009. Towards real-time community detection in large networks. *Phys. Rev. E* 79 (Jun 2009), 066107. Issue 6. doi:10.1103/PhysRevE.79.066107

[25] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for new hardware platforms. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 302–313. doi:10.1109/ICDE.2013.6544834

[26] Renjie Liu, Yichuan Wang, Xiao Yan, Haitian Jiang, Zhenkun Cai, Minjie Wang, Bo Tang, and Jinyang Li. 2025. DiskGNN: Bridging I/O Efficiency and Model Accuracy for Out-of-Core GNN Training. *Proc. ACM Manag. Data* 3, 1, Article 34 (Feb. 2025), 27 pages. doi:10.1145/3709738

[27] Ziyang Liu, Chaokun Wang, Yunkai Lou, and Hao Feng. 2023. Fast Unsupervised Graph Embedding via Graph Zoom Learning. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. 2551–2564. doi:10.1109/ICDE55515.2023.00196

[28] Yong-Shang Long, Zhen Jia, and Ying-Ying Wang. 2018. Coarse graining method based on generalized degree in complex network. *Physica A: Statistical Mechanics and its Applications* 505, C (None 2018), 655–665. doi:10.1016/j.physa.2018.03.080

[29] Yunkai Lou and Chaokun Wang. 2019. OSMAC: Optimizing Subgraph Matching Algorithms with Community Structure. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 1750–1753. doi:10.1109/ICDE.2019.00189

[30] Yunkai Lou and Chaokun Wang. 2024. A Generalized Community-Structure-Aware Optimization Framework for Efficient Subgraph Matching in Social Network Analysis. *IEEE Transactions on Computational Social Systems* 11, 2 (2024), 2545–2557. doi:10.1109/TCSS.2023.3303476

[31] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph Hellerstein. 2010. GraphLab: a new framework for parallel machine learning. In *Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence* (Catalina Island, CA) *(UAI'10)*. AUAI Press, Arlington, Virginia, USA, 340–349.

[32] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. USENIX Association, Santa Clara, CA, 133–148. https://www.usenix.org/conference/fast16/technical-sessions/presentation/lu

[33] 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 Twelfth European Conference on Computer Systems* (Belgrade, Serbia) *(EuroSys '17)*. ACM, NY, USA, 527–543. doi:10.1145/3064176.3064191

[34] 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*. 363–374. doi:10.1109/ICDE.2015.7113298

[35] Grzegorz Malewicz, Matthew H. Austern, Aart J.C 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* (Indianapolis, Indiana, USA) *(SIGMOD '10)*. ACM, NY, USA, 135–146. doi:10.1145/1807167.1807184

[36] Maria Massri, Zoltan Miklos, Philippe Raipin, and Pierre Meye. 2022. Clock-G: A temporal graph management system with space-efficient storage technique. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. 2263–2276. doi:10.1109/ICDE53745.2022.00215

[37] Kiran Kumar Matam, Gunjae Koo, Haipeng Zha, Hung-Wei Tseng, and Murali Annavaram. 2019. GraphSSD: graph semantics aware SSD. In *Proceedings of the 46th International Symposium on Computer Architecture* (Phoenix, Arizona) *(ISCA '19)*. ACM, NY, USA, 116–128. doi:10.1145/3307650.3322275

[38] METIS. 2025. https://github.com/KarypisLab/METIS (last checked on 2025-09).

[39] Dingheng Mo, Junfeng Liu, Fan Wang, and Siqiang Luo. 2025. Aster: Enhancing LSM-structures for Scalable Graph Database. *Proc. ACM Manag. Data* 3, 1, Article 12 (Feb. 2025), 26 pages. doi:10.1145/3709662

[40] MySQL. 2025. https://github.com/mysql/mysql-server (last checked on 2025-10).

[41] Neo4j. 2024. https://github.com/neo4j/neo4j (last checked on 2024-04).

[42] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Inf.* 33, 4 (June 1996), 351–385. https://doi.org/10.1007/s002360050048

[43] Tarikul Islam Papon, Taishan Chen, Shuo Zhang, and Manos Athanassoulis. 2024. CAVE: Concurrency-Aware Graph Processing on SSDs. *Proc. ACM Manag. Data* 2, 3, Article 125 (May 2024), 26 pages. doi:10.1145/3654928

[44] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. 2007. Near linear time algorithm to detect community structures in large-scale networks. *Phys. Rev. E* 76 (Sep 2007), 036106. Issue 3. doi:10.1103/PhysRevE.76.036106

[45] Giulio Rossetti and Rémy Cazabet. 2018. Community Discovery in Dynamic Networks: A Survey. *ACM Comput. Surv.* 51, 2, Article 35 (Feb. 2018), 37 pages. doi:10.1145/3172867

[46] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The network data repository with interactive graph analytics and visualization. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence* (Austin, Texas) *(AAAI'15)*. AAAI Press, 4292–4293.

[47] 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* (Monterey, California) *(SOSP '15)*. ACM, NY, USA, 410–424. doi:10.1145/2815400.2815408

[48] 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* (Farminton, Pennsylvania) *(SOSP '13)*. ACM, NY, USA, 472–488. doi:10.1145/2517349.2522740

[49] Jifan Shi, Biao Wang, and Yun Xu. 2024. Spruce: a Fast yet Space-saving Structure for Dynamic Graph Storage. *Proc. ACM Manag. Data* 2, 1, Article 27 (March 2024), 26 pages. doi:10.1145/3639282

[50] 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* (Xi'an, China) *(ASPLOS '17)*. ACM, NY, USA, 237–251. doi:10.1145/3037697.3037748

[51] Keval Vora, Guoqing Xu, and Rajiv Gupta. 2016. Load the edges you need: a generic I/O optimization for disk-based graph processing. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference* (Denver, CO, USA) *(USENIX ATC '16)*. USENIX Association, USA, 507–522.

[52] Chaokun Wang, Binbin Wang, Bingyang Huang, Shaoxu Song, and Zai Li. 2021. FastSGG: Efficient Social Graph Generation Using a Degree Distribution Generation Model. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. 564–575. doi:10.1109/ICDE51399.2021.00055

[53] Meng Wang, Chaokun Wang, Jeffrey Xu Yu, and Jun Zhang. 2015. Community detection in social networks: an in-depth benchmarking study with a procedure-oriented framework. *Proc. VLDB Endow.* 8, 10 (June 2015), 998–1009. doi:10.14778/2794367.2794370

[54] Songyao Wang, Chaokun Wang, Fang Niu, and Cheng Wu. 2025. LSM-Community: A Graph Storage System Exploiting Community Structure in Social Networks . In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*. IEEE Computer Society, Los Alamitos, CA, USA, 3575–3588. doi:10.1109/ICDE65448.2025.00267

[55] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. 2016. Speedup Graph Processing by Graph Ordering. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. ACM, NY, USA, 1813–1828. doi:10.1145/2882903.2915220

[56] Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, and Ion Stoica. 2013. GraphX: a resilient distributed graph system on Spark. In *First International Workshop on Graph Data Management Experiences and Systems* (New York) *(GRADES '13)*. ACM, NY, USA, Article 2, 6 pages. doi:10.1145/2484425.2484427

[57] Song Yu, Shufeng Gong, Qian Tao, Sijie Shen, Yanfeng Zhang, Wenyuan Yu, Pengxi Liu, Zhixin Zhang, Hongfu Li, Xiaojian Luo, Ge Yu, and Jingren Zhou. 2024. LSMGraph: A High-Performance Dynamic Graph Storage System with Multi-Level CSR, In SIGMOD 2025. *Proc. ACM Manag. Data* 2, 6, Article 243, 28 pages. doi:10.1145/3698818

[58] Wei Zhang, Cheng Chen, Qiange Wang, Wei Wang, Shijiao Yang, Bingyu Zhou, Huiming Zhu, Chao Chen, Yongjun Zhao, Yingqian Hu, Miaomiao Cheng, Meng Li, Hongfei Tan, Mengjin Liu, Hexiang Lin, Shuai Zhang, and Lei Zhang. 2024. BG3: A Cost Effective and I/O Efficient Graph Database in Bytedance. In *Companion of the 2024 International Conference on Management of Data* (Santiago AA, Chile) *(SIGMOD '24)*. ACM, NY, USA, 360–372. doi:10.1145/3626246.3653373

[59] Yu Zhang, Xiaofei Liao, Xiang Shi, Hai Jin, and Bingsheng He. 2018. Efficient Disk-Based Directed Graph Processing: A Strongly Connected Component Approach . *IEEE Transactions on Parallel & Distributed Systems* 29, 04 (April 2018), 830–842. doi:10.1109/TPDS.2017.2776115

[60] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A Computation-Centric Distributed Graph Processing System. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 301–316. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/zhu

[61] 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. *Proc. VLDB Endow.* 13, 7 (mar 2020), 1020–1034. doi:10.14778/3384345.3384351

[62] 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)*. USENIX Association, Santa Clara, CA, 375–386. https://www.usenix.org/conference/atc15/technical-session/presentation/zhu

Received October 2025; revised January 2026; accepted February 2026