SMDG: Enhancing In-Memory Dynamic Graph Processing With Storage-Class Memory

Tongfeng Weng, Mo Sha, Xu Zhou, Jingjing Lu, Wentao Huang, Kenli Li, *Senior Member, IEEE*, and Kian-Lee Tan

***Abstract*—In-memory dynamic graph processing faces three critical challenges: limited DRAM capacity, inefficient concurrent update/query handling, and vulnerability to crashes. Traditional segment-level systems struggle with write amplification on emerging Storage-Class Memory (SCM), while existing persistent-memory systems suffer from coarse-grained synchronization and high recovery overhead. This study presents the Storage-Class Memory Dynamic Graph (SMDG) processing framework, an architecture-level redesign centered on the block as the atomic unit across storage, concurrency, and recovery. The system addresses these challenges through three key innovations. First, a block-granular storage design organizes adjacency data at fixed-size block granularity on heterogeneous DRAM-SCM architecture, employing buffered batched writes to significantly reduce write amplification while preserving logarithmic update complexity. Second, block-level multi-version concurrency control maintains timestamped block versions under per-vertex read-write synchronization to provide task-ordered snapshot visibility for concurrent queries without copying entire vertices or pages. Third, a block-granular crash recovery protocol with decentralized per-vertex logs enables independent parallel reconstruction, ensuring application-level semantic consistency while achieving substantially faster recovery than sequential approaches. Experimental results validate that this unified block-granular design improves update efficiency, sustains mixed update-query workloads with controlled memory overhead, and accelerates crash recovery compared with prior dynamic graph systems.**

***Index Terms*—Dynamic graph, MVCC, packed-memory array, storage-class memory.**

I. Introduction

LARGE-SCALE graph processing faces significant challenges from graph size [1], skewed topology [2], [3], and irregular access patterns [4]. Many applications such as social networks [5], [6], e-commerce [7], and financial systems [8] require dynamic graph processing for real-time analytics [9]. In-memory solutions remain prevalent [10], [11] for latency-sensitive scenarios, with extensive research on representation [2], [12], [13], indexing [14], query processing [15], [16], [17], [18], [19], [20], [21], [22], and transaction management [23].

Despite these advances, three fundamental challenges continue to impede the development of practical large-scale dynamic graph systems. **First**, limited main memory capacity remains a critical bottleneck for scalability. Although modern servers can be configured with hundreds of gigabytes of RAM, this capacity is often insufficient for real-world graphs with billions of edges, and provisioning high-memory machines incurs prohibitive costs for many deployments. **Second**, most in-memory dynamic graph frameworks lack native support for concurrent updates and queries. They typically adopt a ping-pong strategy that alternates between update and query phases via dual buffers, introducing latency that is incompatible with real-time analytics requirements. **Third**, purely in-memory designs are inherently vulnerable to system failures. A crash necessitates reloading historical data and replaying accumulated updates, which not only interrupts ongoing query processing but can also create irreversible backlogs in high-velocity streaming environments.

We present SMDG, a framework leveraging Storage-Class Memory [24], [25] (SCM) for large-scale dynamic graph processing. SCM offers up to 8× DRAM capacity (e.g., 512 GB vs. 64 GB per DIMM) at lower cost, with byte-addressable access [26]. Hybrid DRAM-SCM architectures enable larger graphs through intelligent data placement [27].

However, SCM exhibits lower performance than DRAM, with asymmetric read-write characteristics and susceptibility to write amplification—critical concerns for irregular graph workloads [28], [29]. While superior to disk-based approaches [30], SCM's unique properties necessitate a fundamental rethinking of graph data structures and system architecture rather than straightforward adaptation of existing in-memory designs.

To address these challenges, SMDG adopts a domain-specific architectural approach centered on a novel data structure: Linked Packed Memory Blocks (LPMB). LPMB serves as the foundational component for dynamic graph representation in heterogeneous DRAM-SCM architectures, specifically designed to exploit the strengths of each memory tier while mitigating their respective limitations. Inspired by the Packed Memory Array [31], [32] (PMA)—a dynamic structure that maintains ordered elements in contiguous memory (Section II-C)—LPMB introduces several architectural innovations tailored for SCM

Received 25 February 2025; revised 13 March 2026; accepted 18 March 2026. Date of publication 23 March 2026; date of current version 1 May 2026. The work of Tongfeng Weng and Kian-Lee Tan was supported in part by the Ministry of Education (Title: inPMdb: An in-Persistent Memory Database System and in part by WBS under Grant A8000082-00-00. Recommended for acceptance by X. Yu. *(Corresponding authors: Mo Sha; Jingjing Lu.)*

Tongfeng Weng, Wentao Huang, and Kian-Lee Tan are with the School of Computing, National University of Singapore, Singapore 117417 (e-mail: tf.weng@nus.edu.sg; huang@comp.nus.edu.sg; tankl@comp.nus.edu.sg).

Mo Sha is with Alibaba Cloud, Singapore 189554 (e-mail: shamo.sm@alibaba-inc.com).

Xu Zhou, Jingjing Lu, and Kenli Li are with the College of Computer Science and Electronic Engineering, Hunan University, Hunan 410082, China (e-mail: zhxu@hnu.edu.cn; lujingjing000@hnu.edu.cn; lkl@hnu.edu.cn).

Digital Object Identifier 10.1109/TKDE.2026.3676514

environments. Unlike traditional balanced trees such as B+trees, LPMB exhibits three distinctive characteristics:

- It uses a more flexible balancing strategy, allowing dynamic choices between immediate rebalancing and deferred writes with batching, which lowers memory fragmentation during frequent updates.
- It performs bulk and periodic rebalancing with subtree-level optimization instead of local node fixes, improving cache line alignment and reducing I/O amplification during structural updates.
- It maintains amortized $O(\log N)$ time for edge insertions, deletions, and updates, even with delayed writes and arbitrary operation sequences.

Building on these design principles, SMDG's architecture integrates LPMB with two complementary mechanisms to address the remaining challenges of concurrency and fault tolerance. LPMB distributes adjacency data across fixed-size, cache-aligned SCM blocks while maintaining metadata in DRAM. To enable concurrent query and update operations, the system employs multi-version concurrency control (MVCC [33]) with Copy-on-Write [34] under per-vertex read-write synchronization, so that queries observe a task-ordered snapshot while updates version only the touched adjacency blocks. Additionally, SMDG leverages SCM's non-volatility to implement efficient crash recovery mechanisms that ensure data consistency across system failures.

In summary, SMDG represents an *architecture-level redesign* that fundamentally rethinks dynamic graph processing for Storage-Class Memory (SCM). **Unlike prior dynamic graph systems** based on *segment-level* PMA structures or *page/object-level* persistence mechanisms, SMDG **re-architects** the entire system at the *block level*, proposing an *architectural unification* centered on the *block as the atomic unit* across storage, concurrency, and recovery. This unified design addresses the three principal challenges—limited DRAM capacity, inefficient concurrent access, and crash vulnerability—through coordinated innovations in adjacency management, multi-version concurrency control, and crash-consistent recovery, all operating on a *heterogeneous DRAM–SCM hierarchy* where the block serves as the consistent abstraction enabling system-wide co-design.

This architectural philosophy is realized through the following three main technical contributions.

- *Block-Granular Adjacency Data Management and Write Amplification Mitigation:* We **propose** Linked Packed Memory Blocks (LPMB) with Block-based Memory Management (BMM), shifting maintenance granularity from *segment-level* (conventional PMAs) to *block-level* (fixed-size, cache-aligned 256 B blocks) on *heterogeneous DRAM–SCM hierarchy*. Through *non-contiguous block allocation*, *delayed rebalancing*, and *DRAM-buffered batched writes*, LPMB significantly reduces write amplification while preserving amortized $O(\log N)$ complexity (Theorem 1).
- *Block-Level Multi-Version Concurrency Control for High-Concurrency Access:* We **design** *block-granular MVCC + Copy-on-Write (CoW)* that maintains *per-block version chains* (`next_blk_ptr`) under per-vertex read-write synchronization. This design provides task-ordered snapshot visibility at the same *block-level granularity* as LPMB, avoiding both *page-level* copying and *per-element* version overhead while preserving a precise and implementable concurrency model.
- *Semantic Consistency Guarantee via Block-Level Crash Recovery Protocol:* We **introduce** a **dual-log recovery mechanism** (per-vertex redo log + global block redo log) enabling *block-granular independent recovery*, where each vertex autonomously reconstructs its adjacency blocks without global coordination, ensuring *semantic consistency* while enabling *massively parallel reconstruction* achieving $O(|V| \cdot t/T)$ vs. $O(|E| \cdot t)$ for sequential WAL (Theorem 2).

II. Preliminaries

A. In-Memory Graph Processing Framework

Graph processing frameworks must facilitate efficient updates and queries [35], [36]. In-memory approaches [37] load raw graph data into working memory and transform it into memory-efficient structures for fast processing. However, lack of system-level optimizations can lead to out-of-memory errors and excessive latency when handling large datasets [38]. For extensive graph maintenance operations, frameworks must integrate fault-tolerance mechanisms. Since reconstructing large-scale graph data is time-consuming, in-memory graph frameworks can integrate persistent redo logs to enable efficient recovery [39].

B. Storage-Class Memory (SCM)

Recent advances in byte-addressable Storage Class Memory (SCM) bridge the gap between DRAM and traditional storage, offering much higher capacity per server and lower cost per gigabyte, with faster access than block-based storage [24], [40]. Technologies such as 3D XPoint, MRAM, and ReRAM provide DRAM-like access with persistence. NVDIMM combines non-volatile memory with DRAM to preserve data on power loss. Despite these benefits, SCM is generally slower than DRAM, and write operations require careful management to mitigate write amplification [27], [41]. SCM enables memory tiering, dynamically placing it between DRAM and traditional storage to minimize latency and enhance throughput, particularly beneficial in large-scale analytics and real-time applications. Ensuring data persistence in SCM requires cache line flushes (e.g., CLFLUSHOPT, SFENCE) [39], necessitating careful design for consistent recovery after crashes.

C. Graph Storage Structure

Existing graph storage structures, such as adjacency lists and compressed sparse rows (CSR) [22], [42], are shown in Fig. 1(b, c). In these illustrations, green slots indicate elements that must be shifted after inserting a new edge. These structures involve numerous scattered writes to SCM for managing dynamic graph data, leading to inefficiencies.

To mitigate this, several studies [2], [12], [43], [44] have explored using packed memory arrays (PMA) for dynamic graph

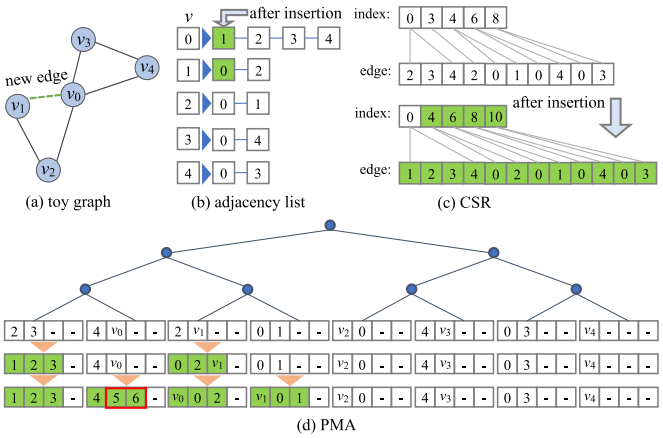

Fig. 1. Examples of dynamic graph representation.

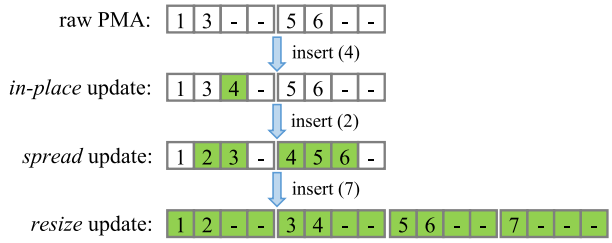

Fig. 2. PMA maintenance operations.

storage. PMA, a mutable variant of CSR, includes empty slots to avoid complete array shifts during insertions. The array is divided into k contiguous segments, each containing $O(N/k)$ slots, where N is the total capacity.

As shown in Fig. 1(d), PMA organizes the array using a binary tree. For a segment (i.e., a window corresponding to a segment at level i covers the leaf segments rooted at the segment) at height i (with leaves at height 0), density thresholds ρ_i and τ_i ensure an amortized update complexity of $O(\log N)$. When a segment's density falls out of the range $[\rho_i, \tau_i]$, a *rebalance* operation redistributes elements within the subtree. Fig. 2 shows core PMA operations. Here, each leaf segment stores four elements, density thresholds $\rho_i = 0.25$ and $\tau_i = 0.75$. Inserting element 4 is handled in place due to enough available slots in the first leaf segment (i.e., the density is 0.75, which is not larger than $\tau_i = 0.75$). After inserting element 2, elements $1, 2, 3, 4$ are stored in the first leaf segment, and the density of the first leaf segment is increased to 1. This requires redistribution across the window consisting of the two leaf segments. Specifically, we need to *rebalance* these six elements across the window (i.e., elements $1, 2, 3$ are stored in the first leaf segment and elements $4, 5, 6$ are placed in the second leaf segment). The final density of the two segments is 0.75. Adding element 7 increases the density to $7 \div 8 = 0.875 > \tau_i = 0.75$, which forces resizing for more slots. We need to allocate two more leaf segments and *rebalance* these seven elements across the four leaf segments. More details on PMA maintenance can be found in [31], [45].

Traditional PMAs, however, struggle in high-latency SCM environments due to inefficient write management. As Fig. 2 shows, *spread* and *resize* operations are primary sources of increased write workload. Segment-based maintenance can also lead to full-array resizing, causing data relocations even for unchanged segments, thus escalating SCM write costs. For instance, inserting elements 5 and 6 into v_0's neighbor list requires spreading across four segments, affecting neighbors of v_1 and v_2. Furthermore, variable segment ranges during spreads complicate MVCC snapshot generation, hindering consistent memory management and data backup in SCM.

III. SMDG Design and Implementation

A. System Overview

We first clarify several key terms used throughout this section. **Storage-Class Memory (SCM)** refers to byte-addressable, non-volatile memory (e.g., Intel Optane PMM [41]) with larger capacity than DRAM but higher write latency and limited endurance [26]. **Block-Granular** denotes operating at fixed-size memory block granularity (e.g., 256 B cache-aligned blocks) rather than individual edges or segments—the unifying principle across SMDG's storage, concurrency, and recovery. **MVCC** (Multi-Version Concurrency Control [33]) maintains timestamped versions for concurrent visibility; in SMDG, block-granular MVCC versions entire adjacency blocks via Copy-on-Write while readers and writers are synchronized by per-vertex read-write locks. **Snapshot Consistency** means each query observes a consistent graph state as of its start time. **Semantic Consistency** refers to application-level graph validity after crash recovery (edges in both adjacency lists, correct degrees, no partial operations).

SMDG targets efficient, scalable dynamic graph processing on a hybrid DRAM–SCM architecture. DRAM-only systems struggle with capacity and cost, while SCM (e.g., Intel Optane PMM) offers larger, cheaper memory but with higher write latency and limited endurance. Rather than simply adapting existing graph systems to this new memory technology, SMDG represents an *architecture-level redesign* that co-designs storage, concurrency, and recovery around a unified abstraction: the *block as atomic unit* across all subsystems. This design bridges the performance–capacity gap and enables reliable, concurrent graph operations.

Fig. 3 illustrates SMDG's architecture organized around a heterogeneous DRAM–SCM hierarchy with three core modules unified by the *block* as the atomic unit:

1) Data placement: Vertex metadata in DRAM (IDs, degrees, pointers), adjacency lists in SCM via LPMBs; task queue dispatches update/query tasks to worker threads.

2) LPMB + BMM: Each vertex's adjacency list stored in cache-aligned, non-contiguous SCM blocks (Section III-B); BMM manages block allocation; buffered rebalancing curbs write amplification.

3) Block-granular MVCC: CoW creates versioned block snapshots (Section III-D); queries traverse version chains under per-vertex read-write synchronization to obtain the correct snapshot view; garbage collection reclaims obsolete snapshots.

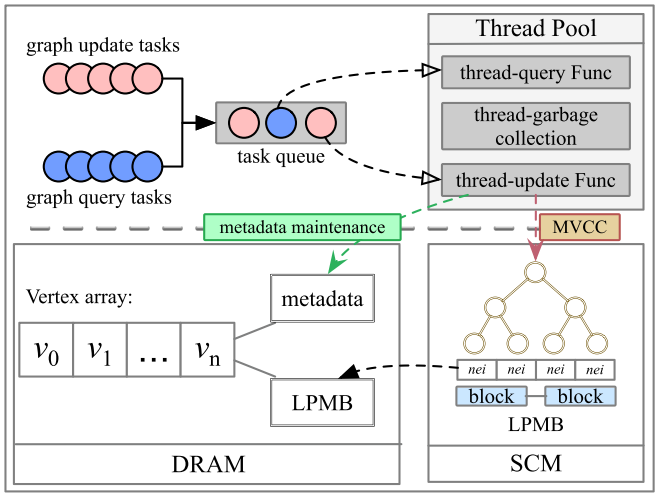

Fig. 3. Architecture overview of SMDG.

4) Block-level recovery: Per-vertex redo logs + block redo logs (Section III-C); independent vertex recovery enables parallel reconstruction. This unification—LPMB organizes blocks, MVCC versions blocks, recovery reconstructs blocks—reduces complexity and ensures consistency via a single version chain structure serving both concurrency and durability.

B. In-Memory Dynamic Graph Representation

1) Block-Based Memory Management (BMM): SMDG employs Block-based Memory Management (BMM) to align memory operations with SCM's cache-line granularity. BMM maintains a DRAM-resident pool of fixed-size blocks for fast allocation and recycling. During updates, SMDG applies changes in DRAM and flushes them in batches to SCM, reducing write amplification. When resizing is needed, new blocks are allocated from the pool and may be non-contiguous in SCM, improving flexibility under frequent updates.

2) Metadata on DRAM: Following semi-external graph models [27], SMDG stores vertex metadata in DRAM while edges reside in SCM. Each vertex maintains: (1) degree counters for frequent access efficiency; (2) pointers to its LPMB blocks in SCM; and (3) a read-write lock for concurrent query/update isolation.

3) Linked Packed Memory Blocks on SCM: Conventional PMAs, when naively ported to SCM, trigger *segment-level rebalancing* that writes multiple cache lines (256 B each on Intel Optane PMM) on every density violation, rapidly exhausting SCM's limited write endurance [26], [46]. This fundamental mismatch between *segment-level* maintenance granularity and SCM's cache-line write granularity creates severe write amplification. To address this problem, we **propose Linked Packed Memory Blocks (LPMB)**, an *SCM-aware, block-granular adjacency storage design* that fundamentally shifts the maintenance granularity from segments to *fixed-size, cache-aligned blocks (256 B)*. LPMB organizes adjacency data at *block-level granularity* via *non-contiguous block allocation* in SCM linked through DRAM pointers, decoupling logical adjacency order from physical SCM layout. Combined with *delayed rebalancing* and *DRAM-buffered batched writes*, this *block-granular* design significantly reduces write amplification while preserving PMA's amortized $O(\log N)$ complexity—with correctness formally proven in Theorem 1.

Different from prior work using a single global PMA [43], SMDG stores each vertex's sorted neighbors in an SCM-resident LPMB, where (-1) marks empty slots and positive integers denote neighbors. LPMB is inspired by PMA [31], [45], retaining its update logic and amortized $O(\log N)$ guarantees. A key feature is the *spread* operation, which redistributes valid elements across allocated blocks when densities drift, keeping updates cheap and avoiding frequent large reorganizations. The interleaved layout enables easy reuse of invalidated slots and minimizes compaction, reducing write amplification on SCM and allowing in-place updates with deferred reorganization. Coupled with BMM, SMDG further cuts SCM writes, improving PMA maintenance efficiency. Given the slot in PMA, SMDG maps it to a physical address on SCM by

$$B_{index} = L_{index}/block\_size, \tag{1}$$

$$B_{offset} = L_{index}\%block\_size, \tag{2}$$

where L_{index} is the index of the slot in PMA, B_{index} indicates the index of the block in the blk_list belonging to the vertex metadata, and B_{offset} is the offset in the block. For an edge update, SMDG identifies the appropriate slot in the source vertex's neighbor LPMB to insert or delete the edge. It then performs a rebalance operation—either spread or resize—to maintain the PMA's density constraints, as illustrated in Fig. 2.

Optimization for spread: During the $spread$ process, which involves replicating all elements within a window adhering to PMA's density constraints, SMDG allocates a temporary array in DRAM to evenly store these elements. Subsequently, this temporary array is migrated to SCM in block sizes. This approach achieves a reduction in write operations to SCM on the one hand, while on the other hand, the replication at the block size level helps mitigate the effects of write amplification.

Optimization for $resize$: PMA maintenance recurses until all segment densities fall within thresholds. If no suitable window exists up to the root, the array is doubled and elements are evenly redistributed (*resize*). To minimize SCM writes during *resize*, SMDG computes the required number of blocks, allocates them from the DRAM-managed pool (creating more via BMM if needed), then evenly redistributes elements across blocks—flushing in aligned batches.

These optimizations collectively enable SMDG to minimize SCM writes during PMA maintenance. In summary, the *block-granular* LPMB design delivers three principal advantages:

i) Write efficiency: By shifting from *segment-level* to *block-level* maintenance with delayed rebalancing and DRAM-buffered batched writes, LPMB significantly reduces SCM write traffic compared to traditional segment-level PMAs.

ii) Formal guarantees: Preserves amortized $O(\log N)$ update complexity with correct rebalancing under arbitrary operation sequences (Theorem 1).

iii) Architectural unification: LPMB's *block as atomic unit* abstraction naturally supports *block-level* snapshots for MVCC and *block-granular* redo logs for crash recovery, creating a cohesive system design where storage, concurrency, and recovery operate on the same fundamental unit.

Theorem 1: Consider a PMA P_{array} with root upper density τ_0, capacity C, and N elements. When inserting a new element such that $N+1 > C \times \tau_0$, the capacity is increased to $2C$ slots. By evenly distributing the elements in P_{array} across all slots based on their indices from largest to smallest, the density of each segment will not exceed τ_0.

Proof: We establish the theorem by analyzing two extreme cases of element distribution: left-aligned and right-aligned configurations, representing scenarios with the highest potential segment densities after redistribution.

Case 1. Left-Aligned Distribution: Suppose the $\lfloor \frac{N}{2} \rfloor + 1$ original elements and the new insertion are stored in the leftmost $\frac{C}{2}$ slots. Following the method described in [31], [45], we first pack all elements to the left and then redistribute them evenly across the expanded capacity of $2C$ slots. The total number of elements is $N+1$, so the average density after redistribution is

$$\text{Density} = \frac{N+1}{2C}.$$

Since $N+1 > C \times \tau_0$, we have

$$\text{Density} = \frac{N+1}{2C} > \frac{C \times \tau_0}{2C} = \frac{\tau_0}{2}.$$

However, because the capacity has doubled, the density in any segment remains below τ_0:

$$\text{Density} = \frac{N+1}{2C} < \tau_0.$$

Case 2. Right-Aligned Distribution: Suppose the elements are stored in the rightmost $\frac{C}{2}$ slots, leaving the leftmost slots empty. Upon doubling the capacity to $2C$, we redistribute the elements evenly across all slots. The average density is the same as in Case 1:

$$\text{Density} = \frac{N+1}{2C} < \tau_0.$$

This ensures that no segment exceeds the root upper density τ_0 after redistribution. In both cases, by evenly distributing the elements across the expanded capacity, the density of each segment remains below τ_0. □

Example: Fig. 4 illustrates the graph storage mechanism within SMDG. As shown in Fig. 4(a), all vertices are arranged in an array. Each vertex's metadata comprises the vertex ID, degree, and a block list housing pointers to SCM blocks that are used to store its neighbors. Additionally, each block contains essential attributes such as the block pointer (start address), next block pointer (used for snapshot backup), and q_stamp (denoting the current snapshot version). The neighbors of a vertex are stored in an LPMB within SCM. Fig. 4(b) showcases the BMM strategy. A memory pool comprises several vacant blocks. For instance, let's consider vertex v_0 with neighbors $\{1, 2, 3\}$ stored in $Block{-}0$ with four slots. Assuming the LPMB segment size matches the block size and the density thresholds $\rho_i = 0.25$ and $\tau_i = 0.75$,

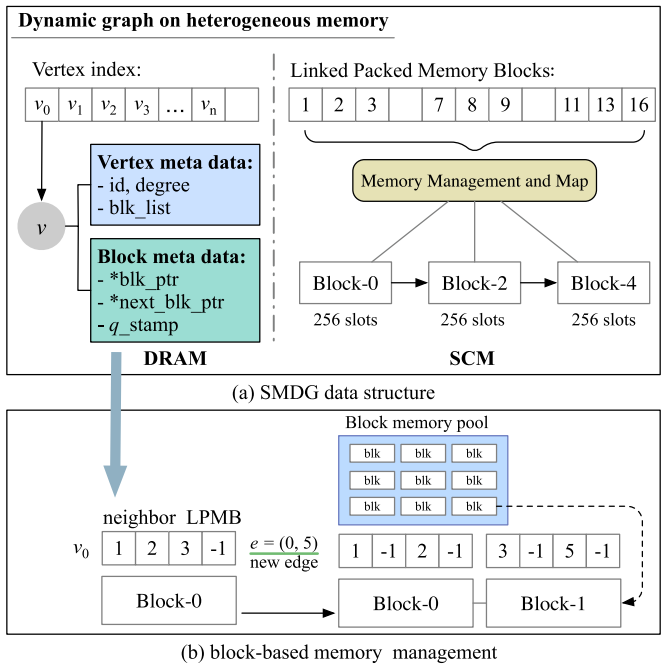

Fig. 4. Data structure and Block-based memory management.

inserting edge $e = (0, 5)$ into v_1 prompts SMDG to allocate and link a new block to $Block{-}0$ (i.e., the *resize* operation). The elements are then evenly distributed across the two blocks. Now, if we insert edge $e = (0, 6)$, the neighbor ID 6 can be directly placed in the fourth slot of the second block, as the density does not exceed 0.75 and SCM supports byte-addressable read/write operations without the need to cache the block in DRAM. After that, we insert edge $e = (0, 4)$. Since the second block is full, it needs to borrow a slot from the first block. In this case, both blocks are first cached into DRAM, where the *spread* operation is performed (i.e., to distribute the valid elements across the two blocks evenly). The updated data is then written back to SCM in a batched manner, reducing data shifting overhead and improving bandwidth utilization. For edge deletion, we set the value of the corresponding slot to -1. If the overall ratio of valid elements falls below a threshold $\rho_i = 0.25$, a *resize* operation is triggered to reduce the number of blocks and reclaim the freed blocks to the block memory pool.

4) Parallel Graph Maintenance: In SMDG, all the vertex metadata is stored in DRAM, and each vertex's attributes, such as degree, concurrent mutex, and neighbor LPMB, are managed individually. The LPMBs associated with different vertices occupy distinct physical regions within SCM. Given that all LPMBs acquire and release blocks from the same block memory pool, SMDG ensures the memory pool is managed atomically, allowing SMDG to handle LPMBs concurrently to enhance its efficiency and parallelism capabilities [19], [21].

C. Crash-Consistent Mechanism

Conventional persistent systems rely on global consistency protocols creating recovery bottlenecks: 2 PC serializes updates, WAL replay is single-threaded and proportional to log size

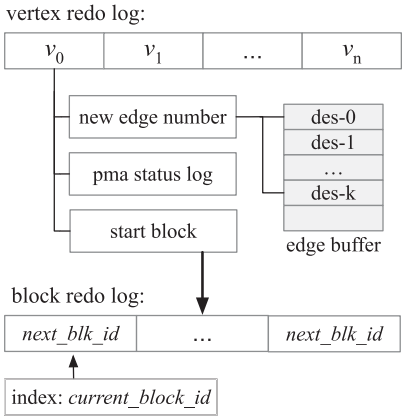

Fig. 5. Redo log for system recovery.

(minutes-to-hours for large graphs [2]), and full-graph reconstruction from checkpoints becomes infeasible for high-velocity streams. To address these limitations, SMDG introduces **block-granular crash consistency** as an *architectural co-design* with LPMB's *block-level* storage and MVCC's *block-level* versioning: decentralizing recovery metadata to per-vertex redo logs and per-block redo logs (CoW snapshot chains) on SCM, making the *adjacency block the atomic recovery unit*. This ensures *semantic consistency*—application-level graph validity where every edge (u, v) appears in both adjacency lists (or neither), vertex degrees match neighbor counts, and no partial operations exist. Crucially, each vertex's recovery is independent (disjoint logs in SCM), enabling *massively parallel reconstruction* achieving $O(|V| \cdot t_{\text{vertex}}/T)$ recovery time with T threads vs. $O(|E| \cdot t_{\log})$ for sequential WAL replay—with correctness formally guaranteed by Theorem 2. This *block as atomic unit* abstraction unifies recovery boundaries with LPMB's storage blocks and MVCC's version blocks, creating a cohesive crash-consistent architecture.

Due to the non-volatile nature of SCM, where data persists even during a system crash, SMDG incorporates the above crash-consistent mechanism to enable fast data recovery. As illustrated in Fig. 5, SMDG maintains two types of redo logs in SCM: a *vertex redo log* and a *block redo log*. For a vertex, the redo log includes an edge buffer, $new\ edge\ number$, $start\ block$, and $pma\ status\ log$. The $new\ edge\ number$ indicates the number of valid new edges in the edge buffer that must be processed during graph recovery. The $start\ block$ represents the block ID linking to the start block backup of the neighboring LPMB. The $pma\ status\ log$ tracks the element count of the LPMB when it was last resized. Using this element count, SMDG recalculates the LPMB parameters, including capacity, segment size, and density threshold, which are then used to process edges in the edge buffer. The block redo log is structured as an array, where the index corresponds to the current block ID, and the value is initialized to -1. When the edge buffer of a vertex is full, SMDG generates backups for all blocks hosting its neighboring LPMB. The $start\ block$ is set to the ID of the first backup block. If multiple blocks are involved, subsequent block IDs are recorded in the block redo log. Once all logs and backup blocks are persisted to SCM, the $new\ edge\ number$ is reset to 0.

Upon a crash, SMDG performs *block-granular independent recovery* by examining each vertex's redo log independently. The recovery process leverages the **dual-log mechanism**: (1) *Per-vertex redo log* determines whether edges are fully persisted ($new\_edge\_number = 0$) or require rollback and replay ($new\_edge\_number > 0$). (2) *Block redo log* provides *block-level CoW snapshots* for reconstructing pre-update LPMB state via the snapshot chain starting from $start\_block$. This *decentralized recovery metadata* design ensures each vertex can independently reconstruct its adjacency block chain and replay buffered edges without global coordination, guaranteeing *semantic consistency* (edges in both adjacency lists, correct degrees, no partial operations). Importantly, SMDG enables *massively parallel recovery:* multiple threads concurrently recover disjoint vertices, achieving $O(|V| \cdot t_{\text{vertex}}/T)$ recovery time—as validated by experimental results in Section IV.

Theorem 2: The crash-consistent mechanism ensures accurate data restoration.

Proof: When SMDG activates its crash-consistent mechanism, it first records each new edge added to or removed from the graph G in the vertex redo log (or edge buffer) on the SCM. Then, SMDG updates the corresponding LPMB with this new edge. If a crash occurs during this process, SMDG can restore the entire graph by using the vertex and block redo logs. This is feasible because the data in the block redo log remains unchanged during LPMB maintenance. In addition, if a crash occurs during the backup generation process, SMDG ensures that both the LPMB backup blockchain and the edge buffer are preserved before the new backup is committed to the SCM. If there is an interruption while generating a new LPMB backup, SMDG can still fully restore the graph from the logs, because the original backup blockchain and edge buffer remain unmodified. □

D. Concurrent Graph Processing

Conventional locking in concurrent graph processing creates bottlenecks: global locks serialize access, fine-grained locks risk deadlock and hub-vertex contention, and readers may starve under write-heavy workloads. Multi-Version Concurrency Control (MVCC) [33] addresses this through timestamped versions that preserve *snapshot consistency*—each query observes a consistent graph state as of its start time (query-stamp), which is critical for iterative graph algorithms (e.g., PageRank) requiring a stable topology. In SMDG, readers and writers are synchronized by per-vertex read-write locks, while block-version chains preserve snapshot visibility. The design goal is to align version management with the SCM-oriented block layout. Existing MVCC implementations suffer from granularity mismatches: *page-level* MVCC in persistent DBMSs copies data at a coarse granularity, while *per-element* versioning in prior graph systems incurs high metadata overhead. We therefore **design block-granular MVCC + Copy-on-Write (CoW)**, an *SCM-aware concurrency protocol* that versions only the touched adjacency blocks at the same *block-level granularity* as LPMB's storage layout. By maintaining *per-block version chains* via `next_blk_ptr`

pointers, SMDG provides task-ordered snapshot visibility while avoiding full-vertex or page-level version copies.

The key technical components and implementation details are presented below.

1) Worker threads: SMDG uses worker threads to manage multiple queries and updates simultaneously. When the number of tasks surpasses the available workers, SMDG places the excess tasks in a waiting queue.

2) Task Allocation. SMDG first classifies each task as an update (insertion/deletion) or a query, then dispatches it to the corresponding routine. Each query receives a unique task ID used as a ***query-stamp*****to mark its start for block snapshots. To preserve a task-ordered snapshot, updates scheduled earlier in the same stream complete before that query begins, while later updates proceed by creating CoW versions of the affected blocks.**

3) Read-Write Lock: SMDG employs a read-write lock on each vertex to isolate concurrent queries and updates. This allows multiple query threads to read a vertex's neighbors simultaneously while ensuring only one update thread can modify them at a time.

4) Query-stamp Block Snapshot: SMDG uses blocks as the fundamental unit for generating snapshots, enabling batch version filtering during neighbor traversal. For example, with 256-byte blocks and 4-byte slots, each block manages up to 64 neighbor versions.

To accommodate edge updates arriving at different query start times, SMDG employs a Copy-on-Write (CoW) mechanism to create block snapshots based on query-stamps. Specifically, before inserting an edge into a block in the LPMB, SMDG first generates a snapshot of the block, assigns it the next query task identifier (query-stamp), and then applies the edge update on the LPMB.

5) Query Access Pattern: When a query task accesses a specific block of a vertex's neighbor LPMB, it needs to search for the appropriate snapshot based on the next block pointer ($next\_blk\_ptr$). If a block snapshot is found where the query-stamp is greater than the query ID, that version replaces the original block in the LPMB. If the query-stamp is not greater than the query ID when $next\_blk\_ptr$ is empty, then the original block in the LPMB is accessed.

6) Garbage Collection: In SMDG, a background thread continuously monitors invalid blocks. Once the task with the earliest start time completes, the thread reclaims block snapshots created prior to it. Since edge updates are directly applied to the neighboring LPMB, which consistently maintains the most recent neighbor versions, all snapshots can be safely discarded upon task completion, eliminating the need for their integration into the LPMB.

Example: Fig. 6 provides a visual representation of the concurrent graph processing methodology. SMDG maintains a neighbor LPMB for each vertex, partitioned into four segments distributed across two SCM blocks. The colored rectangles indicate incoming edge updates. Prior to the initiation of $query{-}1$, a new edge is scheduled for insertion into $seg{-}1$ on $blk{-}0$. To accommodate this update, SMDG generates a snapshot of $blk{-}0$, assigning it the label $q_1{-}stamp$. During the execution of $query{-}1$, because the $next_blk_ptr$ associated with $q_1{-}stamp$ is NULL and the snapshot's label does not exceed the task ID of $query{-}1$, the query accesses the original version of $blk{-}0$ within the LPMB. In a similar manner, SMDG creates snapshots $q_2{-}stamp$ and $q_3{-}stamp$ for $query{-}2$ and $query{-}3$, respectively. Notably, during the execution of $query{-}2$ on $blk{-}1$, since the $next_blk_ptr$ of $q_2{-}stamp$ is not NULL and the label of the subsequent snapshot, $q_3{-}stamp$, exceeds the task ID of $query{-}2$, the access to the neighbors on $blk{-}1$ must occur via the $q_3{-}stamp$ snapshot upon its completion. Following the completion of $query{-}1$, the $q_1{-}stamp$ snapshot is released.

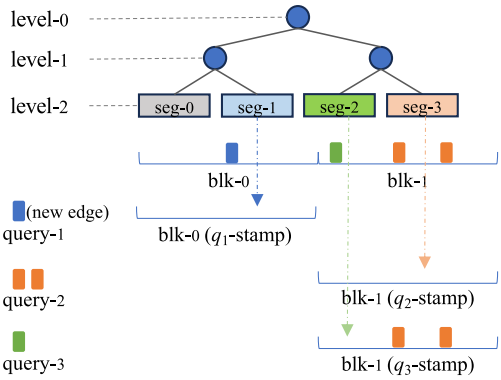

Fig. 6. A block snapshot example.

Algorithm 1: Block-Granular MVCC Operations.

```
1  Procedure UPDATE(vertex v, block B, edge e, timestamp T_current)
2      Acquire write lock on v ;
3      T_min ← GetMinActiveQueryStamp() ;
4      if T_min < T_current then
5          B' ← AllocateBlock() ;        // CoW snapshot
6          Copy content from B to B' ;
7          B'.q_stamp ← T_current ;
8          B'.next_blk_ptr ← B ;
9          Link B' to version chain ;
10     Apply edge update e to B ; // in-place update
11     Release write lock on v ;
12 Procedure QUERY(vertex v, block B, query-stamp T_i)
13     Acquire read lock on v ;
14     B_current ← B ;
15     while B_current ≠ NULL do
16         if B_current.q_stamp ≤ T_i then
17             break ;          // found appropriate
                                   version
18         B_current ← B_current.next_blk_ptr ;
19     Read neighbors from B_current ;
20     Release read lock on v ;
```

Algorithm 1 formalizes the block-granular MVCC workflow: the UPDATE procedure performs CoW when active queries exist, creating snapshot blocks with timestamped metadata linked via version chains; the QUERY procedure traverses the version chain to find the appropriate block version based on query-stamp, guaranteeing snapshot consistency with traversal cost proportional to the chain length.

TABLE I
GRAPH DATASETS

Graph	Vertices	Edges
orkut (OR)	3,072,441	234,369,798
dimacs10-uk (UK)	105,153,952	6,603,753,128
graph500-22 (g500-22)	2,396,657	128,311,470
graph500-26 (g500-26)	32,804,978	2,103,845,706
uniform-22 (uni-22)	2,396,657	128,311,470
uniform-26 (uni-26)	32,804,978	2,103,845,706

Block-granular MVCC with copy-on-write improves performance over naive lock-based or coarse-grained versioning designs through three aspects. First, it preserves **task-ordered snapshot visibility**: readers acquire shared per-vertex locks and writers acquire exclusive per-vertex locks, while version chains ensure that each query sees the appropriate historical block version rather than a partially updated block. Second, it **avoids coarse-grained copying**: SMDG versions only the touched adjacency blocks instead of whole vertices, pages, or entire graph snapshots. Third, it **reduces version metadata overhead**: block-level versioning amortizes copy-on-write across many neighbors in a block rather than per element, lowering metadata and space overhead compared with element-level MVCC [23]. Together, this block-granular MVCC with copy-on-write aligns with LPMB's block-based storage, versions blocks through compact chains instead of edges or pages, and provides a precise block-based concurrency model that complements storage and recovery in a unified design.

E. Implementation Details

We implemented SMDG in C++. Although Optane PMM's memory management provides valuable support for coordinating DRAM and SCM usage [46], its complex and rigid allocation mechanism poses significant challenges in dynamic graph management scenarios. These scenarios typically involve frequent, fine-grained insertions and deletions of edges and vertices, which require highly flexible and low-latency memory allocation strategies. However, Optane's default memory management is optimized for large, contiguous memory regions and lacks native support for lightweight block-level allocation and deallocation. Without additional customization or middleware support, this limitation can lead to high memory fragmentation, increased latency, and reduced performance when handling dynamic updates in large-scale graphs. Therefore, SMDG opts for a custom memory management design tailored to the unique demands of dynamic graph workloads.

During initialization, SMDG creates file-backed memory by generating a temporary file in the SCM directory. Whenever a new vertex is added, SMDG creates a $Vertex$ object for its metadata and allocates an SCM block for its neighbor LPMB. The memory pool is composed of SCM blocks, while a DRAM vector stores access information. SMDG uses $pthread$ [47] worker threads, with one thread responsible for reclaiming unused block snapshots and the remaining threads handling graph operations.

The non-volatile SMDG takes advantage of SCM's persistence to support post-crash recovery. It enforces crash consistency by maintaining three memory-mapped files in SCM for vertex redo logs, block redo logs, and memory blocks. To maintain consistency after crashes, SMDG relies on extra recovery methods, as explained in Section III-C. During recovery, vertex redo logs are loaded in parallel to rebuild the graph efficiently.

IV. EVALUATION

A. Experimental Setup

Evaluation Platform: The experiments are conducted on a server with an Intel Xeon Gold 6326 CPU (2.90 GHz), featuring 2 sockets, each with 16 cores and 2 threads per core. The memory system features a heterogeneous architecture, consisting of 8 x 32 GB DRAM and 8 x 128 GB Intel Optane PMM per socket. Optane PMM is configured in $AppDirect$ mode, serving as an expansion to DRAM.

Graph kernel: To assess query performance in SMDG and to emulate scenarios mixing multiple updates with multiple queries, we evaluated graph algorithms: PageRank (PR) [17], [48], breadth first search (BFS) [18], [20], [49], connected components (CC) [50], and betweenness centrality (BC) [51]. The implementation was adopted from the GAP Benchmark Suite [52].

Datasets: Our experiments are evaluated on two real-world and four synthetic simple graphs in Table I. OR was crawled from the Orkut website and may thus be incomplete. $dimacs10{-}UK$ (U.K.) is the hyperlink network of the $.UK$ domain for the United Kingdom. These two datasets can be downloaded from http://konect.cc/networks/ [53]. The Graph500-x datasets are synthetic power-law graphs, where the scale factor x specifies the number of vertices and edges. In contrast, the Uniform-x datasets share a similar structure but have a uniform degree distribution. In the case of undirected graphs, each insert or delete of edge (u, v) is treated as two operations (for the "directed" edges (u, v) and (v, u)). This ensures fair comparison across all systems.

Compared frameworks: We verify the advantages and limitations of the proposed SMDG by comparing it with the latest existing dynamic processing frameworks. Specifically, we categorize the comparison into three distinct types: DRAM-only, DRAM-SCM, and SCM-only. Here, DRAM-SCM refers to configurations where the graph data is distributed across both DRAM and SCM storage mediums. SMDG is classified as a DRAM-SCM framework since it stores metadata in DRAM while edges are housed in SCM. It is important to note that, in contrast to a DRAM-only environment, our SMDG primarily relies on SCM for data storage, which leads to increased latency for read and write operations. While this may appear disadvantageous for SMDG from a physical perspective, the experimental results clearly highlight the efficiency of SMDG in dynamically managing graph data.

1) DRAM-only Teseo [12]: Teseo is a library for the real-time analysis of dynamic graphs. It is based on a novel variant of the B+ tree, the fat tree, and can translate the vertex identifiers

into a logical domain, easing the adoption of existing algorithms originally developed for static graphs. All vertices and edges are stored in the leaves of a fat tree. A leaf is organized as a PMA. We deploy Teseo in a DRAM-only environment and compare the performance of graph maintenance with SMDG.

2) SCM-only PMA-based storage [31], [43], [45]: As discussed in Section II, PMA has been validated by multiple works as a data structure suitable for dynamic graphs. Based on PMA [31] and GPMA [43], we implemented a dynamic graph storage framework in C++ for SCM applications, named PMAG. In PMAG, all vertices and edges are stored in a unified PMA. Each slot in PMA consists of two integers representing the value and type, respectively.

3) DRAM-only Stinger [54]: Stinger is adjacency list-like. Neighbors of a given vertex are stored in a linked list of edge blocks (fixed 14 edges). Each block contains metadata such as the mark of valid edges within the block.

4) DRAM-only Sortledton [23]: Sortledton is a universal graph data structure that applies the adjacency list-like design. It is the state-of-the-art efficient data structure for dynamic graph storage on DRAM. As the unrolled skip list does not need global rebalancing, Sortledton uses it to store the neighbors of each vertex. In contrast to the original unrolled skip list, it keeps edges within blocks sorted.

5) DRAM-only Terrace [2]: Terrace is a graph streaming system that utilizes a hierarchical data structure design to store a vertex's neighbors in varying data structures based on the vertex's degree. It extends the interface introduced by Ligra [37], a leading graph-processing framework. The neighbors of a vertex are organized across three container levels: an in-place update array, PMA, and B-tree.

6) DRAM-SCM DGAP [44]: DGAP is a framework for efficient dynamic graph analysis on persistent memory. It utilizes PMA with extensive new designs for persistent memory. DGAP puts the vertex array in DRAM and the edge array in SCM. To ensure a fair comparison, we excluded the operation of persisting to SCM during the experiments. In addition, the authors have not provided the implementation for edge deletion. Additionally, due to unidentified bugs encountered while attempting to accelerate the process with OpenMP, we were unable to measure the time cost of batch graph ingestion. Thus, we can only evaluate the one-by-one insertion scenario.

Evaluation scope: The insertion-only, mixed-update, and query-only experiments in Section IV-B–Section IV-C measure the baseline efficiency of graph maintenance and graph querying. They are not intended to isolate the overhead of SMDG's MVCC/CoW mechanism, because block-level CoW is activated only when updates overlap with active queries. The mixed-task studies in Section IV-E are therefore the experiments that directly exercise the version-management protocol under concurrent updates and queries.

B. Graph Insertion

In this experiment, we assess the efficiency of graph maintenance by comparing it with existing baseline frameworks. We initiate all frameworks with an empty data structure and treat every edge of the provided graph as a new edge update. Since some baselines (PMAG, DGAP) only support serial updates (one-by-one), we evaluated two insertion modes: batch insertions and one-by-one insertions. All edges will be processed in one batch for the batch insertion task. Each framework will allocate 64 threads to process the entire graph in parallel. For baseline evaluation, we used the open-source framework gfe_driver , into which we integrated several graph processing systems, including Teseo, Stinger, and Sortledton. We leveraged its support for concurrent batch edge updates by configuring the number of threads. When the number of threads is set to 1, the system operates in a one-by-one insertion mode. When the thread count is set to 64, it corresponds to the batch insertions mode. In addition, the baseline systems do not generate new snapshots during graph update evaluations. Instead, it simply measures the total time required to apply a batch of edge updates, without creating multiple committed versions of the graph. Similarly, in our SMDG framework, we do not generate multiple versions for each block during update performance evaluations. This ensures a fair comparison, as both systems focus solely on the cost of updating the graph representation without engaging their respective multi-versioning mechanisms. We initially configure the block size to 256 slots $\times\ 4 = 1024$ bytes. As existing frameworks like Teseo, Stinger, and Sortledton do not support SCM-based memory allocation, these systems run only on DRAM, while SMDG runs on DRAM-SCM.

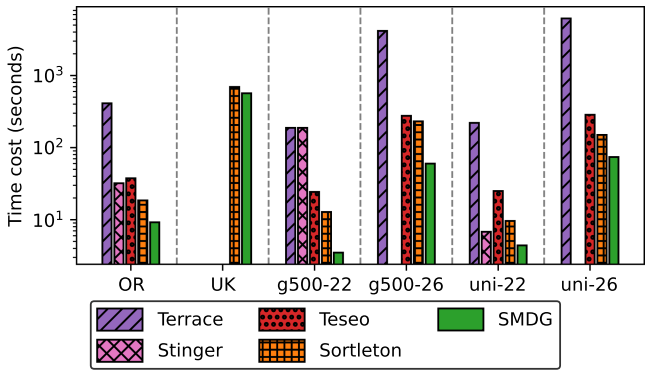

Fig. 7. Evaluation of batch insertions.

The experimental results of batch inserting the whole graph are shown in Fig. 7. Those missing bars indicate that the framework is unable to load such a large graph due to memory limitations or cannot be completed in ten hours. According to the results, SMDG is 44.7x-83.6x faster than Terrace, 1.5x-53.7x as fast as Stinger, 3.9x-6.9x faster than Teseo, and 1.2x-3.9x as fast as Sortledton. For dataset U.K., only Sortledton and SMDG can load the whole graph and the time costs are 694 seconds and 566 seconds, respectively. Although Sortledton performs well overall, Stinger outperformed it on the uni-22 dataset. This indicates that Sortledton cannot consistently maintain high performance across datasets with different distributions, whereas SMDG can. In addition, it is worth noting that SMDG stores all edges in SCM, with higher latency than DRAM, while other frameworks use DRAM exclusively. This demonstrates that SMDG,

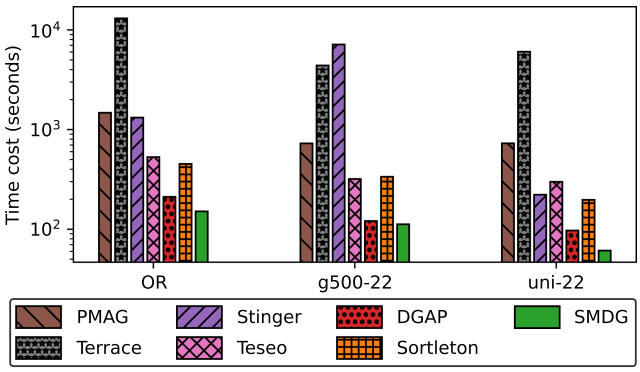

Fig. 8. Evaluation of one-by-one insertions.

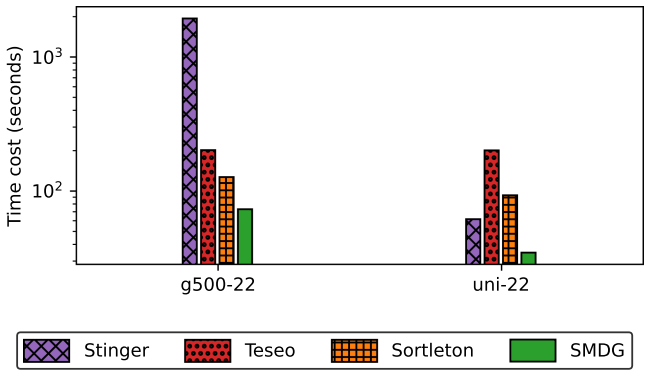

Fig. 9. Evaluation of mixed updates.

though operating on DRAM-SCM hybrid memory, outperforms DRAM-only-storage schemes in managing dynamic graphs.

The one-by-one insertion results are shown in Fig. 8. In this set of experiments, we only evaluate the performance on datasets OR, g500-22, and uni-22 because it will take several tens of hours to process large-scale graphs. As a DRAM-SCM based framework, DGAP outperforms existing baselines except SMDG. This is because DGAP applies an append-only manner while maintaining graphs. New edges will be inserted into the tail of the neighbor list. This approach has poor performance for edge deletion operations on one hand, and on the other hand, neighbors are not ordered (which has been shown to be a crucial attribute in many literatures [2], [12], [23]). Overall, SMDG outperforms all baselines in both batch insertions and one-by-one insertions modes.

C. Graph Update

To evaluate the impact of deletions on the overall throughput while running a balanced mix of edge insertions and deletions with 64 threads (i.e., batch insertion mode), we apply the same setup as in Sortledton [23] and Teseo [12]. Specifically, we generate new datasets from graph500-22 and uniform-22 consisting of 50% edge insertions and 50% edge deletions by using the *graphlog* utility. The final number of edge updates that need to be processed is roughly 10 times the number of edges in the original graph. The insertion edges and deletion edges within the task set are distributed randomly. In this set of experiments, we compare the performance with Stinger, Teseo, and Sortledton, because these three frameworks perform better in processing edge insertions. DGAP did not provide the edge deletion implementation. The total time cost results for maintaining g500-22 and uni-22 are shown in Fig. 9. Stinger performance on g500-22 is poor. The phenomenon is due to the configuration that Stinger holds a fixed size of each edge block. The heavy skew in the g500-22 graph leads to significant performance degradation in memory utilization for Stinger. The results of Teseo and Sortledton are essentially the same across the two datasets. SMDG holds neighbors of each vertex in a LPMB, in which vertex are sorted among different blocks. It can quickly locate the deletion of edges and modify the corresponding slot values through binary search. Additionally, the freed slots can be used for newly inserted edges, enabling efficient handling of mixed updates.

TABLE II
GRAPH KERNEL RUNTIMES

	OR				uni-22			
	PR	BFS	CC	BC	PR	BFS	CC	BC
Terrace	**0.79**	**0.04**	**0.17**	**0.25**	**0.57**	**0.01**	**0.10**	**0.06**
DGAP	0.91	0.08	0.28	0.38	0.88	0.13	0.20	0.53
SMDG	0.96	0.14	1.34	0.45	1.00	0.09	0.76	0.33

D. Graph Query

In this experimental set, we use the four graph kernels (i.e., PR, BFS, CC, and BC) to verify the performance of SMDG in processing graph querying tasks. For BFS and BC, the vertex with the maximum degree will be set as the start vertex. As SMDG puts all edges on SCM, we compare the graph analysis performance with DGAP, in which edges are also stored on SCM. To evaluate the impact of higher SCM latency on graph queries, we also conducted a comparison with the DRAM-only framework Terrace, which is designed on Ligra. The runtime results are shown in Table II. Terrace outperforms the other two DRAM-SCM-based frameworks, partly due to the inherent advantages of the Ligra system and partly because the read latency of DRAM is lower than that of SCM. Since SMDG uses an LPMB at each vertex to store neighbors, retrieving neighbors during traversal requires accessing these blocks and verifying neighbor connections by checking the validity of the slot data. In contrast, DGAP updates vertex neighbors in an append-only manner, enabling contiguous storage in SCM without individually checking slot validity, resulting in greater efficiency than SMDG in graph analysis tasks. As shown in Figs. 7 and 8, SMDG's performance loss in graph analysis is minimal compared to its advantages in graph maintenance. Furthermore, SMDG outperforms existing frameworks in update-intensive tasks.

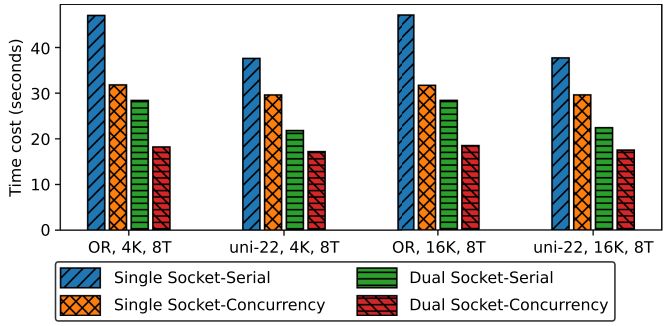

Fig. 10. Evaluation of mixed PageRank and updates.

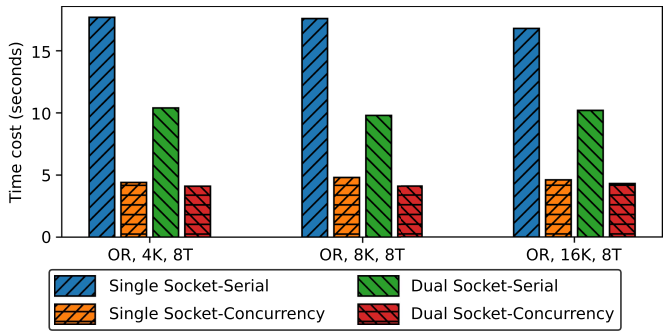

Fig. 11. Self-Evaluation of mixed queries (Connected Components) and updates.

E. Mixed Tasks Processing

As designed in Section III-D, SMDG can process multi-queries and multi-updates concurrently. To exercise this mechanism, we construct streaming task sequences comprising edge insertions, edge deletions, and graph queries. We initiate a sequence of random edge deletions, submit a graph query (PageRank or Connected Components), and repeat this cycle. Throughout the process, queries observe a task-ordered snapshot: updates submitted earlier in the stream complete before that query begins, while later updates proceed by creating CoW versions of the touched blocks so that the running query can continue to read its historical view. Historical blocks are asynchronously reclaimed once they are no longer referenced by active queries. Following two rounds of task submissions, the deleted edges are reinserted into the original graph and interleaved with graph query tasks. This setup encompasses a total of 4 K or 16 K update tasks and 8 query tasks. All tasks are submitted in streaming mode as shown in Fig. 3; updates and queries are therefore active in the system concurrently rather than being executed as two separate batches. We assess concurrent processing on single-socket and dual-socket settings separately. In the single-socket environment, SMDG uses 32 worker threads in total: 16 threads execute updates and 16 threads execute concurrent queries. In the dual-socket environment, SMDG uses 64 worker threads in total: 32 threads execute updates and 32 threads execute concurrent queries.

Figs. 10 and 11 present the runtime results from our evaluation of concurrent processing. The terms "Single Socket-Serial" and "Dual Socket-Serial" refer to the end-to-end execution times of the corresponding updates and PageRank or Connected Components tasks when they are executed without overlapping queries and updates. In contrast, "Single Socket-Concurrency" and "Dual Socket-Concurrency" represent the end-to-end processing times for streaming task batches, including 4 K or 16 K update tasks and 8 query tasks, where SMDG leverages its concurrency mechanism. We report end-to-end runtime here as a system-level view of the benefit of overlapping updates and queries; the update-only and query-only studies earlier in this section remain the primary references for isolated update and query efficiency. The notation "OR, 4 K, 8 T" indicates SMDG processing 4,000 update tasks and 8 queries on the OR dataset. The results clearly demonstrate significant performance improvements with SMDG's concurrency mechanism, regardless of socket configuration. The number of update tasks has minimal impact on overall performance, as SMDG efficiently maintains dynamic graphs, making update time negligible compared to query time. Specifically in Fig. 10, "Single Socket-Concurrency" outperforms "Single Socket-Serial" by 1.3x to 1.5x, while "Dual Socket-Concurrency" shows a 1.3x to 1.6x speedup over "Dual Socket-Serial." Although using multiple sockets may introduce latency due to NUMA node access, the increased CPU cores enhance concurrency, improving both query and update performance. Overall, "Dual Socket-Concurrency" outperforms "Single Socket-Serial" by 2.2x to 2.6x, demonstrating the effective scalability of SMDG's concurrency mechanism in multi-socket environments. As shown in Fig. 11, the results are consistent with previous findings, confirming the effectiveness of the concurrency mechanism and highlighting its scalability under various query workloads.

To further separate update-side and query-side behavior in the mixed-workload setting, we add a multi-metric comparison between SMDG and Sortledton on g500-22 under 16,000 updates (8,000 insertions and 8,000 deletions) with 8, 16, and 24 concurrent queries. We report total runtime, update throughput, query throughput, average query time, and peak DRAM RSS. For compactness, these measurements are summarized in one multi-panel figure, but each panel isolates a different aspect of update-side or query-side behavior. We choose Sortledton here because it supports concurrent mixed updates with both insertions and deletions, making it an appropriate baseline for this setting. We do not include Teseo in this figure because our mixed-workload comparison focuses on baselines for which stable measurements are available under the same deletion-capable configuration.

Fig. 12 shows that as query concurrency increases from 8 to 24, SMDG exhibits only a moderate reduction in update throughput, from 2584.71 ops/s to 1744.76 ops/s, whereas Sortledton drops more sharply from 2961.12 ops/s to 1083.06 ops/s. At the same time, SMDG's query throughput scales from 1.29 to 2.62 qps, while the average query time decreases slightly from 2.81 s to 2.14 s. We attribute this behavior to online reclamation of obsolete versions, which shortens version chains during execution and lowers the traversal cost seen by later queries. In contrast, Sortledton maintains a consistently higher query latency of around 5 s and requires substantially more DRAM, with peak RSS exceeding 10.8 GB compared with SMDG's roughly

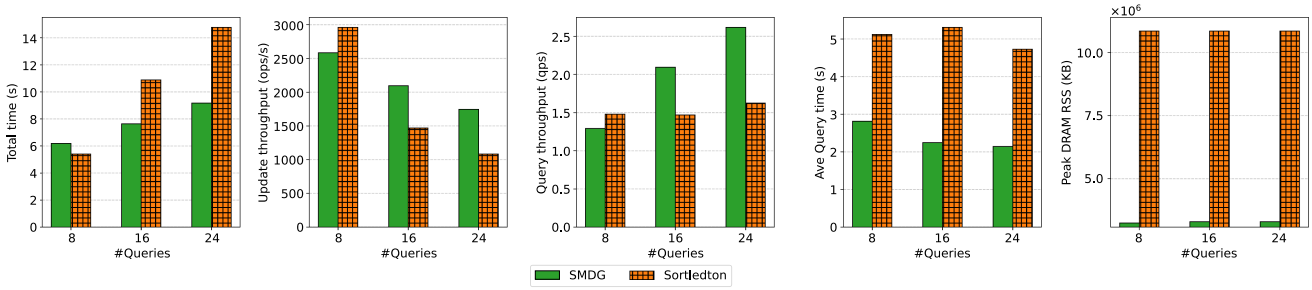

Fig. 12. Multi-metric mixed-workload comparison of SMDG and Sortledton on g500-22 under 16 K updates and varying query concurrency.

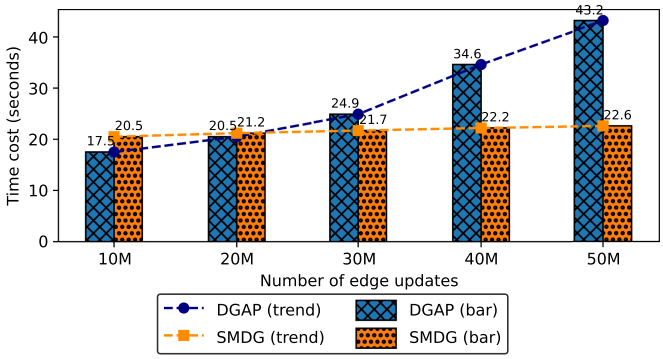

Fig. 13. Evaluation of mixed queries and updates *v.s.* DGAP.

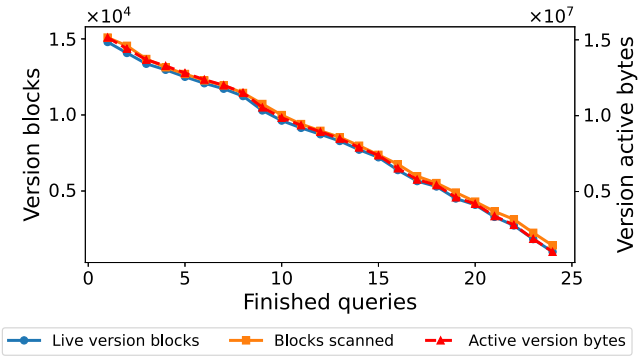

Fig. 14. Version reclamation during concurrent query execution.

3.2 GB. These results show that the block-based MVCC design not only sustains mixed-workload throughput, but also controls memory amplification more effectively under increasing query concurrency.

In addition, we compare our method with DRAM-SCM DGAP under a mixed workload of updates and queries. Since the open-source implementation of DGAP does not support edge deletion, this comparative experiment is limited to insertion-query mixed workloads. We use the same dual-socket worker-thread budget as above (64 worker threads in total) and evaluate both systems on the Orkut dataset by randomly selecting [10 M, 20 M, 30 M, 40 M, 50 M] edges as update tasks and submitting them together with 10 PageRank query tasks. Fig. 13 reports the end-to-end runtime of these mixed workloads. This figure provides an end-to-end mixed-workload comparison rather than a decomposition of update throughput and query latency. Under this setting, DGAP incurs lower overall time cost than SMDG when the update workload is small (10 M and 20 M). However, as the update workload increases, SMDG demonstrates significantly higher efficiency in maintaining dynamic graphs. In terms of trend, SMDG exhibits near-linear growth in runtime as the number of updates increases, while DGAP grows much more steeply. This highlights SMDG's scalability for update-heavy mixed workloads within the insertion-query setting supported by DGAP.

To further analyze the memory cost of block-granular Copy-on-Write during concurrent execution, Fig. 14 tracks three related quantities on g500-22 as queries complete: Blocks scanned, Live version blocks, and Active version bytes. Live version blocks and Active version bytes are nearly proportional because the active memory footprint is determined by the number of live CoW blocks multiplied by the fixed block size, while Blocks scanned reflects the structurally visible version-chain load seen during traversal. As queries progressively finish, all three metrics decline monotonically: live version blocks decrease from 14,795 to 1,002, and active version memory shrinks from approximately 15.1 MB to about 1.0 MB. This shows that although block-level CoW temporarily creates extra versions, the additional space is reclaimed online once snapshots are no longer needed, preventing long-term accumulation of obsolete blocks and reducing traversal overhead for later queries.

Taken together, Figs. 12 and 14 provide two complementary views of the memory tradeoff. The former reports a directly comparable cross-system DRAM metric (peak DRAM RSS), while the latter explains the internal SCM/NVM-side evolution of versioned state during concurrent execution. Because the compared systems place graph state on different memory tiers and expose different memory-accounting interfaces, a single uniform total-runtime-space metric would be difficult to interpret fairly. We therefore report these two views together, along with Fig. 18, to explain the practical memory behavior of SMDG under mixed workloads.

F. Crash Recovery

In this series of experiments, we assess the effectiveness of the crash-consistent mechanism in SMDG. In comparison to traditional in-memory graph processing, which necessitates

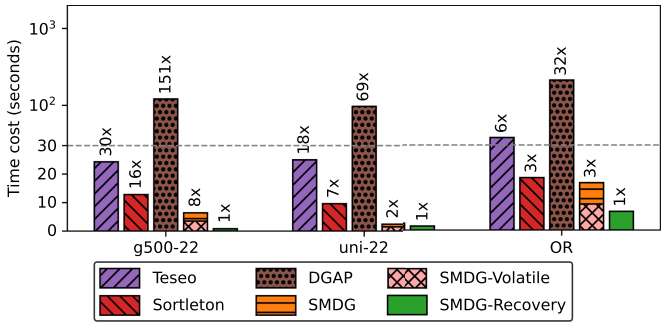

Fig. 15. Evaluation of graph crash recovery.

a complete rebuild of the entire graph representation following a crash, SMDG leverages logs and data blocks on non-volatile SCM to swiftly restore services. We note, as outlined in Section III-C, that this rapid recovery incurs additional overhead due to extra mechanisms, particularly in maintaining redo logs and other corresponding ones. To illustrate this, we introduced an ablation study, termed SMDG-Volatile, which represents the time consumption for rebuilding the graph representation without activating the rapid recovery. Fig. 15 shows that the final bar (SMDG-Recovery) highlights the fast recovery time due to SCM's non-volatile nature, while the others reflect full graph reconstruction times for each method. These comparisons are restricted to datasets that all baselines, including Teseo, Sortledton, and DGAP, can handle.

The results clearly demonstrate that SMDG, under the premise of its inherent advantages in building dynamic graph representations, specifically saves 2x to 8x the time in the Recovery phase compared to rebuilding (SMDG), and this gap widens to 3x to 151x when compared to non-SMDG baselines. This significantly accelerates recovery times after a crash, which is crucial for latency-sensitive dynamic graph analytics. However, it is important to acknowledge the trade-off between the benefits of rapid recovery and the overhead introduced by enabling this mechanism for graph evolution (SMDG vs. SMDG-Volatile), thus necessitating careful consideration of practical application scenarios to optimize the balance between graph maintenance efficiency and recovery speed.

G. Parameter Impact Analysis

This experiment investigates the impact of SMDG's tunable parameters on performance and memory usage, analyzing graph maintenance across different configurations and evaluating memory consumption for block metadata and storage in both DRAM and SCM.

Thread number: We evaluate the impact of thread count on edge insertion performance. Following the setup in Section IV-B, we initialize SMDG with an empty data structure and insert all edges of the given graph. The number of threads limits the concurrency of vertex maintenance in SMDG (see Section III-B4). Thread counts are set to $\{1, 8, 16, 32, 64\}$, with a default block size of 256 slots. Fig. 16 shows that execution time decreases as thread count increases, achieving speedups ranging from 13 to 44 times under 64 threads compared to single-threaded execution. This consistent performance trend across different datasets indicates the effectiveness of parallelization in SMDG.

Block size: We examine the impact of block sizes with varying slots on memory management, setting the number of threads to 1 for sequential edge insertion. In SMDG, each vertex's neighbors are stored in an LPMB, maintained as a PMA distributed across discrete memory blocks. Fig. 17 shows that maintenance times vary less than 10% across different block sizes, indicating robustness in dynamic graph maintenance efficiency regardless of block size.

Memory usage: Although block size does not significantly affect performance, it directly impacts DRAM and SCM memory usage. Larger blocks reduce the number of memory blocks and the associated metadata overhead in DRAM, conserving space. As described in the design of SMDG in Section III-B, each memory block has corresponding metadata stored in the vertex metadata blk_list in DRAM. Fig. 17 illustrates that as the block size increases, the total number of blocks required to store LPMBs decreases. However, since each LPMB requires at least one block, the rate of decrease slows at larger block sizes.

Block size selection: Due to the power-law distribution in real-world data, many vertices have degrees lower than the number of slots in a memory block, risking SCM space waste. Optimal block size balances DRAM and SCM usage. Analyzing block metadata (20 bytes) and slot size (4 bytes), we evaluated DRAM and SCM space utilization under varying block sizes for different datasets. In Fig. 18, the left Y-axis represents DRAM usage, while the right Y-axis indicates SCM usage. Increasing block size reduces DRAM metadata space by $50\% - 72\%$, but increases SCM usage by $55\% - 75\%$. Considering these trends, a block size of 64 to 128 slots (256 to 512 bytes) balances memory utilization.

V. Related Work

A. Static Graph Storage

In the field of graph computing, the selection of a graph storage structure is crucial for enhancing algorithm efficiency and reducing memory consumption. With the continuous expansion of the scale of graph data and the complexity of application scenarios, researchers have developed various graph storage structures to adapt to different needs. The Coordinate format (COO) [55], where each non-zero element is stored as a triplet along with the coordinates of its location in the matrix. CSR (Compressed Sparse Row) is widely used in graph storage. Due to its compact storage method and good support for parallel access, CSR has become the first choice for many high-performance graph computing frameworks [56], [57]. Adjacency lists [58] are one of the most straightforward forms of graph storage, maintaining a list of adjacent nodes for each vertex. Although adjacency lists are not as memory-efficient as CSR, they offer high flexibility, particularly suitable for dynamic modifications of the graph. Under certain conditions, when the graph is relatively dense or efficient edge existence query support is needed, adjacency matrices [59] provide an effective solution. Although this method is less space-efficient compared to the previously discussed approaches, it enables more intuitive and

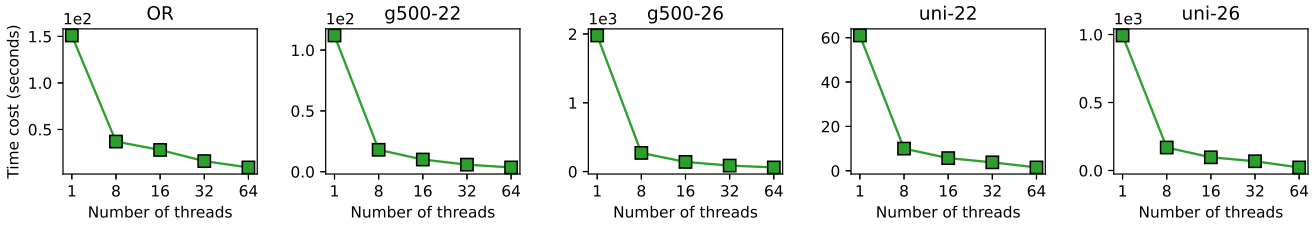

Fig. 16. Evaluation of dynamic graph insertions on different thread numbers.

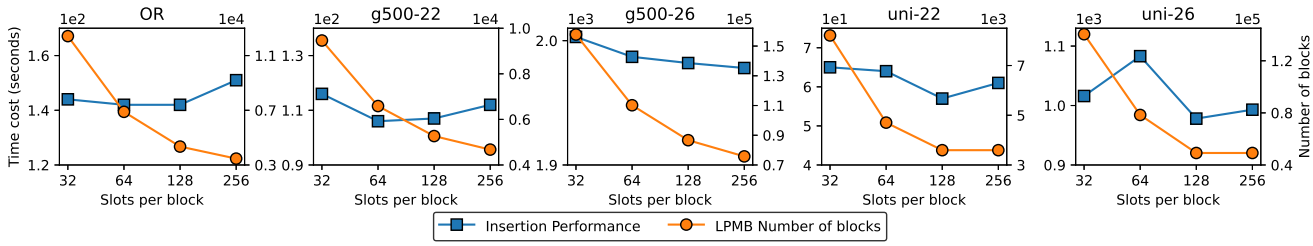

Fig. 17. Evaluation of dynamic graph insertions under different block sizes with the corresponding numbers of blocks.

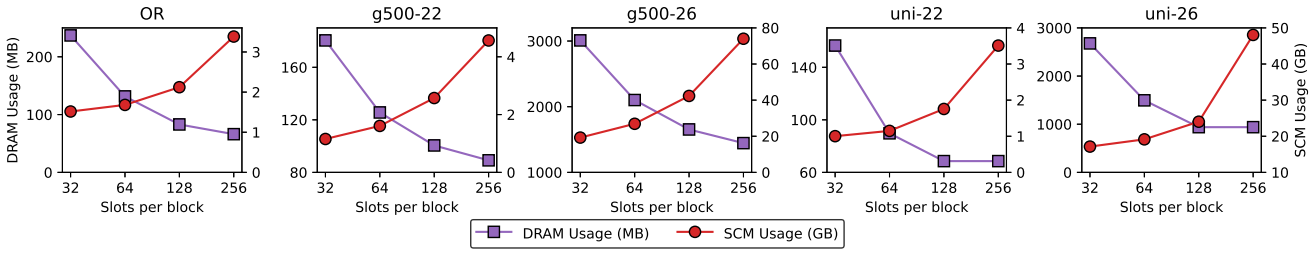

Fig. 18. Evaluation of memory usage on both DRAM and SCM under different block sizes.

straightforward implementations for specific algorithms, such as network flow and path search [60].

B. Dynamic Graph Storage

Dynamic graph storage has received growing attention, as structures designed for static graphs often incur high maintenance overhead under frequent updates. PMA [31], [45] maintains a sorted array and supports efficient insertions and deletions, and Sha et al. [43] extended PMA for dynamic graph storage. Follow-up work explored parallel techniques to accelerate PMA batch updates [61], [62]. These efforts mainly focus on data structure improvements and parallel update strategies, but do not address reducing memory write costs, even though high write latency is a key bottleneck for SCM-based systems. Traditional PMA operates at *segment-level* granularity, where rebalancing requires reorganizing large contiguous segments, causing high write amplification on persistent memory. SMDG instead adopts *block-level* storage using fixed-size, cache-aligned 256 B blocks allocated non-contiguously. This shift from segment-level to block-level maintenance, together with delayed rebalancing and DRAM-buffered batched writes, lowers SCM write traffic while keeping PMA's amortized $O(\log N)$ complexity. In addition, block granularity aligns storage with concurrency control (block-level MVCC) and crash recovery (independent per-block recovery), forming a unified system design rather than an isolated data structure improvement.

C. Dynamic Graph Processing Frameworks

Single-node dynamic graph systems include LLAMA [63], which employs layered representations for dynamic graphs, and Teseo [12], which uses sparse arrays with gaps and a fat-tree index for efficient updates. GraphOne [64] provides an in-memory framework with durability via external non-volatile devices, and XPGraph [35] extends it with designs specialized for persistent memory. Terrace [2] targets streaming graphs with a degree-aware hierarchical layout, while LSGraph [65] improves DRAM-based streaming workloads through locality-centric scheduling. DGAP is designed for dynamic graph analytics on persistent memory using a single mutable CSR structure with persistence-oriented optimizations. These systems either rely on data structures that are not aligned with heterogeneous DRAM–SCM environments (for example, Terrace's B-tree causing high random I/O) or do not handle all update patterns efficiently (for example, DGAP's limited support for high-rate deletions). In contrast to page-level persistence used in XPGraph and DGAP, segment-level versioning in LLAMA, and per-element transactional version management in systems such as Sortledton, SMDG introduces a block-granular unified

design across storage (LPMB fixed-size blocks), concurrency (block-level MVCC with copy-on-write), and recovery (dual-log mechanism). This alignment enables (1) SCM-aware write reduction through delayed rebalancing and DRAM-buffered batched writes that lower write amplification; (2) task-ordered snapshot visibility through per-block version chains coordinated with per-vertex read-write synchronization; and (3) parallel recovery through decentralized per-vertex redo logs, avoiding the sequential WAL replay seen in GraphOne- and XPGraph-style recovery. Our evaluation (Section IV) therefore emphasizes the benefit of aligning storage, versioning, and recovery around the same block abstraction rather than introducing a completely new locking discipline.

VI. Conclusion

This paper presents SMDG, an architecture-level redesign of dynamic graph processing for heterogeneous DRAM-SCM systems. Rather than simply porting existing techniques to new hardware, SMDG introduces an architectural unification centered on the block as the atomic unit across storage, concurrency, and recovery. We propose block-granular adjacency management (LPMB) to mitigate write amplification while preserving logarithmic complexity, block-level multi-version concurrency control with per-vertex read-write synchronization to provide task-ordered snapshot visibility through block version chains, and block-granular crash recovery with decentralized logs to support massively parallel reconstruction with semantic consistency guarantees. These innovations are formally proven and experimentally validated, demonstrating substantial performance advantages over state-of-the-art systems in update throughput, concurrent access, and recovery speed. SMDG represents a cohesive system design where storage layout, concurrency protocol, and recovery mechanism operate on a unified block abstraction, pointing the way forward for persistent-memory graph processing in latency-sensitive, high-concurrency environments.

References

[1] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, "One trillion edges: Graph processing at facebook-scale," *Proc. VLDB Endow.*, vol. 8, no. 12, pp. 1804–1815, 2015.

[2] P. Pandey, B. Wheatman, H. Xu, and A. Buluc, "Terrace: A hierarchical graph container for skewed dynamic graphs," in *Proc. Int. Conf. Manage. Data*, 2021, pp. 1372–1385.

[3] R. Chen, J. Shi, Y. Chen, and H. Chen, "PowerLyra: Differentiated graph computation and partitioning on skewed graphs," in *Proc. 10th Eur. Conf. Comput. Syst.*, 2015, pp. 1:1–1:15.

[4] G. Cong, S. B. Kodali, S. Krishnamoorthy, D. Lea, V. A. Saraswat, and T. Wen, "Solving large, irregular graph problems using adaptive work-stealing," in *Proc. 37th Int. Conf. Parallel Process.*, 2008, pp. 536–545.

[5] R. Cheng et al., "Kineograph: Taking the pulse of a fast-changing and connected world," in *Proc. 7th ACM Eur. Conf. Comput. Syst.*, 2012, pp. 85–98.

[6] W. Han et al., "Chronos: A graph engine for temporal graph analysis," in *Proc. Eur. Conf. Comput. Syst.*, 2014, pp. 1:1–1:14.

[7] J. Yuan et al., "Community trend prediction on heterogeneous graph in E-commerce," in *Proc. ACM Int. Conf. Web Search Data Mining*, 2022, pp. 1319–1327.

[8] X. Zhu et al., "Taking the pulse of financial activities with online graph processing," *ACM SIGOPS Oper. Syst. Rev.*, vol. 55, no. 1, pp. 84–87, 2021.

[9] S. Sakr et al., "The future is big graphs: A community view on graph processing systems," *Commun. ACM*, vol. 64, no. 9, pp. 62–71, 2021.

[10] M. S. Hassan, T. Kuznetsova, H. C. Jeong, W. G. Aref, and M. Sadoghi, "GRFusion: Graphs as first-class citizens in main-memory relational database systems," in *Proc. SIGMOD Conf.*, 2018, pp. 1789–1792.

[11] J. Lin, S. Li, Y. Ding, and Y. Xie, "Overcoming the memory hierarchy inefficiencies in graph processing applications," in *Proc. IEEE/ACM Int. Conf. Comput. Aided Des.*, 2021, pp. 1–9.

[12] D. De Leo and P. Boncz, "Teseo and the analysis of structural dynamic graphs," *Proc. VLDB Endowment*, vol. 14, no. 6, pp. 1053–1066, 2021.

[13] J. Shi, B. Wang, and Y. Xu, "Spruce: A fast yet space-saving structure for dynamic graph storage," *Proc. ACM Manag. Data*, vol. 2, no. 1, Mar. 2024, Art. no. 27.

[14] F. Zhang and S. Wang, "Effective indexing for dynamic structural graph clustering," *Proc. VLDB Endow.*, vol. 15, no. 11, pp. 2908–2920, 2022.

[15] J. Mondal and A. Deshpande, "EAGr: Supporting continuous ego-centric aggregate queries over large dynamic graphs," in *Proc. Int. Conf. Manage. Data*, Snowbird, UT, USA, 2014, pp. 1335–1346.

[16] A. Pacaci, A. Bonifati, and M. T. Özsu, "Evaluating complex queries on streaming graphs," in *Proc. Int. Conf. Data Eng.*, 2022, pp. 272–285.

[17] W. Guo, Y. Li, M. Sha, and K. Tan, "Parallel personalized pagerank on dynamic graphs," *Proc. VLDB Endow.*, vol. 11, no. 1, pp. 93–106, 2017.

[18] M. Sha, Y. Li, and K. Tan, "GPU-based graph traversal on compressed graphs," in *Proc. Int. Conf. Manage. Data*, Amsterdam, The Netherlands, 2019, pp. 775–792.

[19] W. Guo, Y. Li, M. Sha, B. He, X. Xiao, and K. Tan, "GPU-accelerated subgraph enumeration on partitioned graphs," in *Proc. Int. Conf. Manage. Data*, Portland, OR, USA, 2020, pp. 1067–1082.

[20] Q. Lyu, M. Sha, B. Gong, and K. Lyu, "Accelerating depth-first traversal by graph ordering," in *Proc. 33rd Int. Conf. Sci. Stat. Database Manage.*, Tampa, FL, USA, 2021, pp. 13–24.

[21] M. Sha, Y. Li, and K. Tan, "Self-adaptive graph traversal on GPUs," in *Proc. Int. Conf. Manage. Data*, 2021, pp. 1558–1570.

[22] T. Weng et al., "Efficient projection-based algorithms for tip decomposition on dynamic bipartite graphs," *IEEE Trans. Knowl. Data Eng.*, vol. 37, no. 2, pp. 626–640, Feb. 2025.

[23] P. Fuchs, J. Giceva, and D. Margan, "Sortledton: A universal, transactional graph data structure," *Proc. VLDB Endow.*, vol. 15, no. 6, pp. 1173–1186, 2022.

[24] W. Huang, Y. Ji, X. Zhou, B. He, and K.-L. Tan, "A design space exploration and evaluation for main-memory hash joins in storage class memory," *Proc. VLDB Endowment*, vol. 16, no. 6, pp. 1249–1263, 2023.

[25] M. L. Gallo and A. Sebastian, "An overview of phase-change memory device physics," *J. Phys. D: Appl. Phys.*, vol. 53, no. 21, 2020, Art. no. 213002.

[26] A. Van Renen, L. Vogel, V. Leis, T. Neumann, and A. Kemper, "Persistent memory I/O primitives," in *Proc. 15th Int. Workshop Data Manage. New Hardware*, 2019, pp. 1–7.

[27] L. Dhulipala et al., "Sage: Parallel semi-asymmetric graph algorithms for nvrams," *Proc. VLDB Endow.*, vol. 13, no. 9, pp. 1598–1613, May 2020.

[28] X. Zhu, W. Han, and W. Chen, "GridGraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning," in *Proc. USENIX Annu. Tech. Conf.*, 2015, pp. 375–386.

[29] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: Edge-centric graph processing using streaming partitions," in *Proc. 24th ACM Symp. Operating Syst. Princ.*, 2013, pp. 472–488.

[30] Y. Shao, B. Cui, L. Chen, M. Liu, and X. Xie, "An efficient similarity search framework for simrank over large dynamic graphs," *Proc. VLDB Endow.*, vol. 8, no. 8, pp. 838–849, 2015.

[31] M. A. Bender, E. D. Demaine, and M. Farach-Colton, "Cache-oblivious B-trees," in *Proc. 41st Annu. Symp. Foundations Comput. Sci.*, 2000, pp. 399–409.

[32] A. Itai, A. G. Konheim, and M. Rodeh, "A sparse table implementation of priority queues," in *Proc. Int. Colloq. Automata Lang. Program.*, 1981, pp. 417–431.

[33] Y. Wu, J. Arulraj, J. Lin, R. Xian, and A. Pavlo, "An empirical evaluation of in-memory multi-version concurrency control," *Proc. VLDB Endow.*, vol. 10, no. 7, pp. 781–792, 2017.

[34] A. Sharma, F. M. Schuhknecht, and J. Dittrich, "Accelerating analytical processing in MVCC using fine-granular high-frequency virtual snapshotting," in *Proc. SIGMOD Conf.*, 2018, pp. 245–258.

[35] R. Wang, S. He, W. Zong, Y. Li, and Y. Xu, "XPGraph: Xpline-friendly persistent memory graph stores for large-scale evolving graphs," in *Proc. 55th IEEE/ACM Int. Symp. Microarchitecture*, 2022, pp. 1308–1325.

[36] R. Wang, W. Zong, S. He, Y. Li, and Y. Xu, "Scalable and high-performance large-scale dynamic graph storage and processing system," *ACM Trans. Storage*, vol. 21, 2025, Art. no. 18.
[37] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," in *Proc. 18th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2013, pp. 135–146.
[38] G. Malewicz et al., "Pregel: A system for large-scale graph processing," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2010, pp. 135–146.
[39] S. Chen and Q. Jin, "Persistent B^+-Trees in non-volatile main memory," *Proc. VLDB Endow.*, vol. 8, no. 7, pp. 786–797, 2015.
[40] R. F. Freitas and W. W. Wilcke, "Storage-class memory: The next storage system technology," *IBM J. Res. Dev.*, vol. 52, no. 4, pp. 439–447, 2008.
[41] J. Izraelevitz et al., "Basic performance measurements of the intel optane DC persistent memory module," 2019, *arXiv:1903.05714*.
[42] M. Besta, M. Fischer, V. Kalavri, M. Kapralov, and T. Hoefler, "Practice of streaming processing of dynamic graphs: Concepts, models, and systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 34, no. 6, pp. 1860–1876, Jun. 2023.
[43] M. Sha, Y. Li, B. He, and K.-L. Tan, "Accelerating dynamic graph analytics on GPUs," *Proc. VLDB Endow.*, vol. 11, no. 1, pp. 107–120, Sep. 2017.
[44] A. A. R. Islam and D. Dai, "DGAP: Efficient dynamic graph analysis on persistent memory," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, Denver, CO, USA, 2023, pp. 93:1–93:13.
[45] M. A. Bender and H. Hu, "An adaptive packed-memory array," *ACM Trans. Database Syst.*, vol. 32, no. 4, pp. 26–es, 2007.
[46] S. Scargall, *Programming Persistent Memory: A Comprehensive Guide for Developers*. Berlin, Germany: Springer, 2020.
[47] B. Nichols, D. Buttlar, and J. Farrell, *Pthreads Programming: A POSIX Standard for Better Multiprocessing*. Sebastopol, CA, USA: O'Reilly Media, 1996.
[48] M. Bianchini, M. Gori, and F. Scarselli, "Inside pagerank," *ACM Trans. Internet Technol.*, vol. 5, no. 1, pp. 92–128, 2005.
[49] M. Kurant, A. Markopoulou, and P. Thiran, "On the bias of BFS (breadth first search)," in *Proc. 22nd Int. Teletraffic Congr.*, 2010, pp. 1–8.
[50] L. He, X. Ren, Q. Gao, X. Zhao, B. Yao, and Y. Chao, "The connected-component labeling problem: A review of state-of-the-art algorithms," *Pattern Recognit.*, vol. 70, pp. 25–43, 2017.
[51] U. Brandes, "A faster algorithm for betweenness centrality," *J. Math. Sociol.*, vol. 25, no. 2, pp. 163–177, 2001.
[52] S. Beamer, K. Asanović, and D. Patterson, "The gap benchmark suite," 2015, *arXiv:1508.03619*.
[53] J. Kunegis, "Konect: The koblenz network collection," in *Proc. 22nd Int. Conf. World Wide Web*, 2013, pp. 1343–1350.
[54] D. Ediger, R. McColl, J. Riedy, and D. A. Bader, "STINGER: High performance data structure for streaming graphs," in *Proc. IEEE Conf. High Perform. Extreme Comput.*, 2012, pp. 1–5.
[55] K. Kourtis, V. Karakasis, G. Goumas, and N. Koziris, "CSX: An extended compression format for SpMV on shared memory systems," *ACM SIGPLAN Notices*, vol. 46, no. 8, pp. 247–256, 2011.
[56] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "{GraphX}: Graph processing in a distributed dataflow framework," in *Proc. 11th USENIX Symp. operating Syst. Des. Implementation*, 2014, pp. 599–613.
[57] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the GPU," in *Proc. 21st ACM SIGPLAN Symp. Princ. Pract. parallel Program.*, 2016, pp. 1–12.
[58] A. Mhedhbi, P. Gupta, S. Khaliq, and S. Salihoglu, "A indexes: Lightweight and highly flexible adjacency lists for graph database management systems," 2021, *arXiv:2004.00130*.
[59] F. Harary, "The determinant of the adjacency matrix of a graph," *Siam Rev.*, vol. 4, no. 3, pp. 202–210, 1962.
[60] D. Arroyuelo, A. Gómez-Brandón, and G. Navarro, "Evaluating regular path queries on compressed adjacency matrices," in *Proc. Int. Symp. String Process. Inf. Retrieval*, 2023, pp. 35–48.
[61] B. Wheatman and H. Xu, "A parallel packed memory array to store dynamic graphs," in *Proc. Workshop Algorithm Eng. Experiments*, 2021, pp. 31–45.
[62] B. Ramadan, P. Christen, H. Liang, and R. W. Gayler, "Dynamic sorted neighborhood indexing for real-time entity resolution," *J. Data Inf. Qual.*, vol. 6, no. 4, pp. 1–29, 2015.
[63] P. Macko, V. J. Marathe, D. W. Margo, and M. I. Seltzer, "LLAMA: Efficient graph analytics using large multiversioned arrays," in *Proc. IEEE 31st Int. Conf. Data Eng.*, 2015, pp. 363–374.
[64] P. Kumar and H. H. Huang, "Graphone: A data store for real-time analytics on evolving graphs," *ACM Trans. Storage*, vol. 15, no. 4, pp. 1–40, 2020.
[65] H. Qi et al., "LSGraph: A locality-centric high-performance streaming graph engine," in *Proc. 19th Eur. Conf. Comput. Syst.*, 2024, pp. 33–49.

Tongfeng Weng received the master's degree from the School of Information Engineering, Wuhan University of Technology, in 2018, and the doctoral's degree from the College of Computer Science and Electronic Engineering, Hunan University, Changsha, China. He is a research fellow with the School of Computing, National University of Singapore. His research interests include graph computing and parallel computing.

Mo Sha received the PhD degree from the School of Computing, National University of Singapore, in 2020. He is currently a research scientist with Apsara Lab, Alibaba Cloud. His research interests include focuses on leveraging emerging and modern hardware to enhance performance, security, and scalability in data management systems.

Xu Zhou received the master's degree from the College of Computer Science and Electronic Engineering, Hunan University, in 2009. She is currently an associate professor with the Department of Information Science and Engineering, Hunan University, Changsha, China. Her research interests include parallel computing and data management.

Jingjing Lu received the master's degree from the School of Information Engineering, Wuhan University of Technology, in 2018. She is currently working toward the PhD degree with the College of Computer Science and Electronic Engineering, Hunan University. Her research interests include 3D computer vision and deep learning, with a particular focus on 3D shape generation and completion.

Wentao Huang received the BEng degree in electronic engineering from Shandong University, in 2017, and the MEng degree in computer science from the Renmin University of China, in 2020. He is currently working toward the PhD degree with the School of Computing, National University of Singapore (NUS), Singapore. His research interests include database systems and high-performance data systems on emerging hardware.

Kenli Li (Senior Member, IEEE) received the PhD degree in computer science from the Huazhong University of Science and Technology, China, in 2003. He is currently a Cheung Kong professor of computer science and technology with Hunan University, and the vice-principal with Hunan University. His major research interests include parallel and distributed processing. He serves on the editorial board of the *IEEE Transactions on Computers*.

Kian-Lee Tan received the PhD degree in computer science from the National University of Singapore, Singapore, in 1994. He is the dean with the School of Computing, National University of Singapore (NUS), Singapore. His current research interests include query processing and optimization, database performance, data science, and distributed graph computing. He is a member of ACM.