Scalable and High-Performance Large-Scale Dynamic Graph Storage and Processing System

RUI WANG, Zhejiang University, Hangzhou, China and Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security, Hangzhou, China
WEIXU ZONG, Zhejiang University, Hangzhou, China
SHUIBING HE, Zhejiang University, Hangzhou, China and Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security, Hangzhou, China
YONGKUN LI, University of Science and Technology of China, Hefei, China
YINLONG XU, Computer Science, University of Science and Technology of China, Heifei, China

Existing in-memory graph storage systems that rely on DRAM have scalability issues because of the limited capacity and volatile nature of DRAM. The emerging persistent memory (PMEM) offers us a chance to solve these issues through its larger capacity and non-volatile characteristics. However, simply adapting existing DRAM-based graph storage systems to PMEM would result in inefficient PMEM stores and accesses, including high read and write amplification to PMEM, imbalanced work division for PMEM accesses, and costly remote PMEM access across NUMA nodes. These issues severely limit the performance of large graph processing.

In this article, we aim at achieving scalable and high-performance graph processing in PMEM. We first propose an *XPLine-friendly graph storage model* that uses vertex-centric graph buffering, hierarchical vertex buffer managing, and in-place vertex block merging to optimize PMEM graph storage. Furthermore, we develop a *scalable graph processing model* that leverages multi-threaded work dividing and NUMA-friendly graph accessing to optimize PMEM graph accesses. Based on these techniques, we implement XPGraph, a PMEM-based graph storage system for large-scale evolving graphs, and several variants for different system settings. Our experiments demonstrate that XPGraph surpasses the state-of-the-art in-memory graph storage system on a PMEM-based system by 3.07× to 4.99× in update performance and up to 5.87× in query performance, and performs much better in highly parallel multi-threaded scenarios.

CCS Concepts: • **Information systems → Hierarchical storage management**; *Phase change memory;*

Additional Key Words and Phrases: Dynamic graphs processing, persistent memory, graph storage

The work of R. Wang was supported in part by the Science and Technology Program of Zhejiang Province under Grant 2024C01019, Hangzhou Joint Fund of the Zhejiang Provincial Natural Science Foundation of China under Grant LHZSD24F020001, and Zhejiang University Education Foundation Qizhen Scholar Foundation. The work of S. He was supported in part by the National Science Foundation of China under Grant 62172361, the Major Projects of Zhejiang Province under Grant LD24F020012, and the Pioneer and Leading Goose R&D Program of Zhejiang Province under Grant 2024SSYS0002.

Authors' Contact Information: Rui Wang, Zhejiang University, Hangzhou, Zhejiang, China and Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security, Hangzhou, Zhejiang, China; e-mail: rwang21@zju.edu.cn; Weixu Zong, Zhejiang University, Hangzhou, Zhejiang, China; e-mail: zorax@zju.edu.cn; Shuibing He, (Corresponding author) Zhejiang University, Hangzhou, Zhejiang, China and Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security, Hangzhou, Zhejiang, China; e-mail: heshuibing@zju.edu.cn; Yongkun Li, University of Science and Technology of China, Hefei, Anhui, China; e-mail: ykli@ustc.edu.cn; Yinlong Xu, Computer Science, University of Science and Technology of China, Heifei, Anhui, China; e-mail: ylxu@ustc.edu.cn.

ACM 1553-3077/2025/02-ART18
https://doi.org/10.1145/3715332

ACM Reference Format:
Rui Wang, Weixu Zong, Shuibing He, Yongkun Li, and Yinlong Xu. 2025. Scalable and High-Performance Large-Scale Dynamic Graph Storage and Processing System. *ACM Trans. Storage* 21, 2, Article 18 (February 2025), 34 pages. https://doi.org/10.1145/3715332

1 Introduction

Graphs are widely used in many real-world applications, as they can naturally represent the relationships between entities and discover hidden information in these relationships. Graph storage systems are essential for high-performance graph update and analysis, and have attracted wide attention in recent years [18, 38, 58, 59, 71, 72]. The most commonly used in-memory storage formats for evolving graphs include: (1) edge list [69, 70], which stores each edge as a record and can support fast data ingestion for evolving graphs, but suffers from low query performance, and (2) adjacency list [6, 25, 63, 97], which puts all neighbors of a vertex together thus enabling efficient graph queries, but does not benefit fast graph updates. Some recent dynamic graph storage works, such as GraphOne [44], combine the two complementary graph storage formats into a hybrid storage thus supporting efficient graph updates and queries simultaneously.

As graphs grow larger, the scalability of these in-memory graph systems is limited by the machine's DRAM capacity. For instance, GraphOne cannot run on the Yahoo Web [92], a medium-sized graph with 6.6 billion edges, using a server with 128 GB DRAM, due to the out-of-memory error. To handle large graphs, many graph systems resort to distributing large graphs across a cluster of machines [9, 11, 29, 51, 55, 96] or storing them on external disks [26, 40, 46, 49, 66, 78, 82, 98]. However, these systems also suffer from performance issues caused by high communication costs between cluster machines or high I/O costs for internal and external memory interactions. Additionally, to ensure crash consistency, graph updates need to be persisted to the durable disk, which adds extra I/O persistence and recovery costs.

Persistent memory (**PMEM**) is a promising technology that provides non-volatility and enhanced storage capacity, and has gained significant attention with the release of Intel Optane Persistent Memory [60]. Although Intel has discontinued its Optane business recently, we believe progress can still be made for PMEM technology and PMEM-based data storage, as emerging technologies like CXL are expected to bring new opportunities[22]. Intel Optane Persistent Memory leverages 3D-*XP*oint media for internal data storage. Before accessing data in the 3D-XPoint media, memory requests are initially served by a small internal buffer, known as the *XPBuffer*, and subsequently translated into 256-byte data units, referred to as *XPLine*. PMEM offers an alternative approach to solve the scalability problem of in-memory graph stores, and has the potential to achieve high-performance graph storage for large-scale evolving graphs, while also ensuring data persistence.

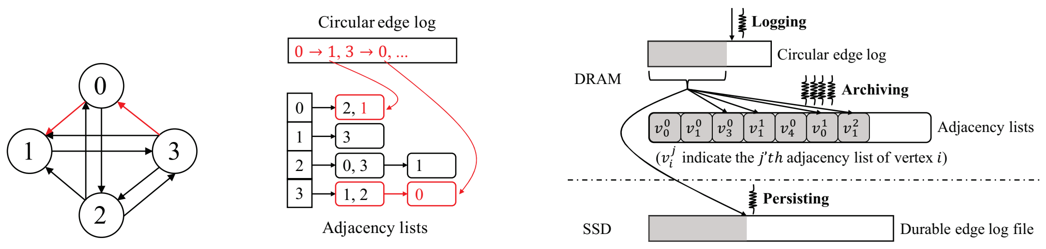
However, simply adapting existing DRAM-based graph systems to PMEM would experience a significant performance drop, due to the completely different performance characteristics between DRAM and PMEM. For example, when we use GraphOne to import the Friendster graph [27], it takes 6.37× longer time on PMEM than on DRAM. We also found that, apart from the small differences in hardware bandwidth between PMEM and DRAM, software designs are the main causes of performance degradation for three reasons. First, DRAM-based graph systems often generate a lot of intensive small random writes, e.g., 4-byte vertex IDs, to different adjacency lists of different vertices. These small random writes have little impact on the performance of DRAM-based systems, because DRAM has high random write performance. However, each 4-byte random write to PMEM may trigger a 256-byte XPLine read-modify-write operation, thus causing serious read and write amplification problems and becoming the performance bottleneck. Second, we need to divide the work among multiple threads for graph updating, but existing solutions would result in severe work imbalance for PMEM accesses, especially in highly parallel multi-threaded scenarios. These unevenly divided edges would cause quite different completion time for different threads in PMEM, making the other threads wait for the slowest one and limiting the system's scalability. Third, because of the limited DIMM slots and cores of a single CPU, multiple CPUs of **non-uniform memory access (NUMA)** architecture are necessary to provide the massive bandwidth and capacity of PMEM [81]. However, the latency of cross-NUMA access of PMEM is much higher than that of DRAM, especially in multi-threaded access scenarios [93], which further worsens the PMEM access efficiency problem in existing in-memory graph systems.

Recently, various efforts have been made to carefully manage the data stored in PMEM and improve the PMEM access efficiency, such as PMEM allocators [5, 7, 20, 52, 61, 68], PMEM indexes [12, 14, 36, 50], PMEM based key-value stores [15, 35, 87], and PMEM based file systems [13, 16, 24, 89]. However, these efforts are not suitable for graph systems, which have different access patterns than the above systems, i.e., highly random accesses and poor data locality. Therefore, in order to efficiently handle large-scale evolving graphs, it is necessary to design a dedicated PMEM-based graph storage system.

An intuitive solution to the read-write amplification problem is to use DRAM as a buffer to cache and merge data updates to PMEM, but this is not easily applied to dynamic graphs due to the following challenges. First, edge updates for evolving graphs have poor data locality, which makes it hard to design an efficient buffering strategy. Second, the data stored in the DRAM buffer would be lost after a power outage, which may cause data inconsistency issues. Third, the large performance gains obtained by the buffering strategy often come with high DRAM space requirements, which may limit system scalability. Moreover, when dealing with dynamic graphs with deleted data, there are also challenges in merging the DRAM-buffered and the PMEM-resident graph data, and recycling the DRAM and PMEM space.

To address the challenges and improve graph storage and processing efficiency, we present XPGraph, an efficient graph storage and processing system for large-scale evolving graphs on PMEM. XPGraph adopts an *XPLine-friendly graph storage model* with an elaborated vertex buffer strategy to minimize PMEM read and write amplification. Additionally, XPGraph develops a *scalable graph processing model* to enhance data processing efficiency in highly parallel multi-threaded scenarios and eliminate remote PMEM accesses across NUMA nodes. Our main contributions are summarized as follows:

— We propose a *vertex-centric graph buffering strategy* by allocating a dedicated DRAM vertex buffer to temporarily cache edge updates for each vertex. We further design a *hierarchical vertex buffer managing scheme* to effectively reduce the DRAM space overhead of these vertex buffers without compromising performance benefits. We also develop an *in-place vertex block merging method* to optimize the graph data deletion performance by a localized block-reuse PMEM allocation, and improve the PMEM space efficiency.
— We design a *multi-threaded work division strategy* to first divide the archiving edges into fine-grained ranges by a self-adaptive strategy, and then allocate these small ranges to multiple threads by an anti-greedy strategy, thus realizing the balanced work division under multi-threaded scenarios. We also introduce a *NUMA-friendly graph accessing method* to completely avoid the remote PMEM accesses across NUMA and improve the graph data accessing performance.
— We implement the PMEM-based prototype system XPGraph and also extend it to three variant systems for better generality, i.e., battery-backed system XPGraph-B, DRAM-only system XPGraph-D, and SSD-based version XPGraph-S. We conduct extensive experiments to demonstrate their efficiency. Results show that XPGraph achieves 3.07× to 4.99× higher



(a) Example graph (b) Hybrid graph storage format (c) Graph ingestion process

Fig. 1. Storage and access process in GraphOne.

ingestion rate as well as up to 5.87× higher query rate, compared with the state-of-the-art in-memory graph storage system implemented on a PMEM-based system. XPGraph also performs much better in higher parallel multi-threaded scenarios.

2 Background and Motivation

We first introduce the storage format and access process of the DRAM-based graph storage strategy. Then we demonstrate the characteristics of the emerging persistent memory (PMEM) and how it differs from the traditional DRAM. Finally, we analyze the limitations of DRAM-based graph storage strategy in supporting PMEM resident evolving graphs.

2.1 DRAM-based Graph Stores

We use the state-of-the-art in-memory evolving graph storage system GraphOne [44] to introduce the storage format and access process of the DRAM-based graph stores. Note that the storage formats and the vertex-centric random access pattern used in GraphOne, are commonly used in many other DRAM-based graph systems like [18, 25, 38, 56, 59, 71, 72]. GraphOne uses a hybrid storage format that combines edge list and adjacency list. GraphOne first uses a circular edge log in the edge list format to store the latest graph updates, which enables efficient graph data ingestion. It then uses many adjacency lists to store the edges that are archived periodically from the edge log, which enables efficient graph queries.

Figure 1(b) shows the hybrid graph storage of the sample graph in Figure 1(a). Figure 1(c) illustrates the graph ingestion process in GraphOne, which consists of several phases. In the *logging phase*, a dedicated logging thread appends the incoming edge updates to the tail of the circular edge log, thus achieving a high ingestion rate. When the number of edges reaches a predefined threshold, GraphOne enters a parallel *archiving phase*, which converts the older edges to adjacency list format, by grouping all incremental neighbors of the same vertex into a single adjacency list, thus supporting efficient graph queries. Specifically, GraphOne adopts a *global batched edge-centric archiving strategy*. It first divides these batched edges into multiple temporary edge lists based on the source vertex IDs of these edge data, and then uses multiple archiving threads to process these temporary edge lists in parallel. For each archiving thread, it first counts the degree increment of each vertex to allocate an adjacency list of an appropriate size, then traverses each edge and appends the neighbor to the corresponding adjacency list. In addition, GraphOne also has a *persisting phase*, which uses a separate persisting thread to periodically write the edge data to a durable edge log file on disk for coarse-grained data persistence.

2.2 Persistent Memory

We use the latest PMEM product, i.e., Intel Optane Persistent Memory 200 Series (hereafter called *Optane*), to demonstrate the access process and the performance characteristics of PMEM-based

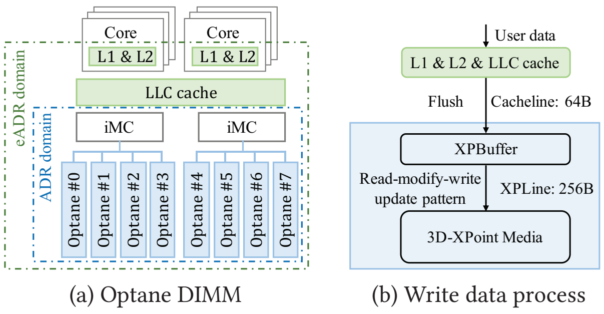


Fig. 2. Intel Optane Persistent Memory 200 Series.

systems. Figure 2(a) shows that Optane sits on the memory bus just like normal DRAM, and communicates with the processor's iMC (integrated memory controller). However, Optane differs from DRAM in its internal access process, as shown in Figure 2(b). (1) Optane uses the 3D-Xpoint media to store data internally, whose actual physical access granularity is 256-byte XPLine. (2) Optane has an internal XPBuffer to cache and merge some data updates, so a data write to the 3D-Xpoint media is actually a read-modify-write with XPLine granularity. (3) Optane has limited ability in handling concurrent accesses from multiple threads. (4) Optane has more severe NUMA (non-uniform memory access) effects than DRAM, especially for cross-NUMA accesses from multiple threads [93]. Therefore, Optane's performance characterization is more complex and can be affected by many factors, such as access type (read vs. write), access pattern (sequential vs. random), access size, and so on [93], and we need to carefully manage the data in Optane.

Note that although Intel has recently discontinued the Optane business, further PMEM and its substitutes, such as CXL-based Memory-Semantic SSD [67], are emerging. The new products may still share some characteristics with Optane, like mismatched access granularity between PMEM and its internal media, limited performance under multiple threads, and cost NUMA effects. Therefore, we believe that PMEM technology and PMEM-based data storage can still make progress, and we can still use Optane as a study case and the evaluation testbed.

2.3 Limitations for PMEM Graph Stores

DRAM-based graph storage systems perform well for DRAM-resident graphs, but the graph scale it can support is limited by the DRAM capacity. For instance, GraphOne fails to run on the medium-sized graph Yahoo Web [92] with 6.6 billion edges by using a server with 128 GB DRAM, due to out-of-memory error (refer to Section 5.2). Moreover, volatile DRAM-based graph stores also need an extra disk for data persistence, which incurs extra durability and recovery costs.

To enable large-scale graph processing in memory with fine-grained consistency guarantee and fast recovery, PMEM with large capacity and non-volatility offers a good opportunity. However, directly applying the storage formats designed for DRAM to real PMEM may lead to significant performance degradation due to the different characteristics between DRAM and PMEM. We implement a PMEM-based GraphOne (called GraphOne-P) by migrating the circular edge log and all adjacency lists from DRAM to PMEM, keeping metadata and intermediate data in DRAM, and then disabling additional persistence operations. We find that GraphOne-P takes nearly five minutes to ingest all 2.6 billion edges in the small Friendster graph. This execution time is 6.37 times that of the original DRAM-based GraphOne (called GraphOne-D).

High read and write amplification in PMEM. Figure 3(a) further shows the logging and archiving performance of GraphOne-D and GraphOne-P separately. The logging process with

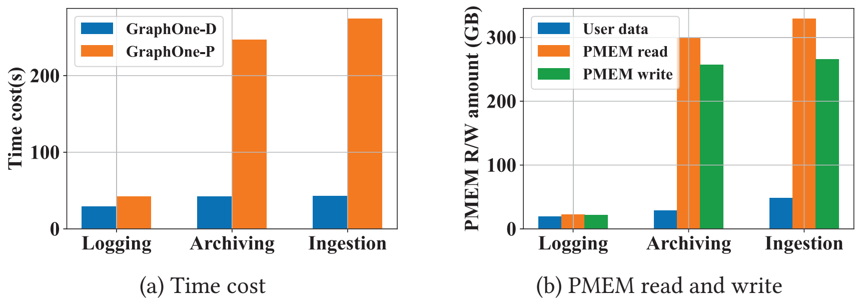


Fig. 3. Compare GraphOne-D with GraphOne-P.

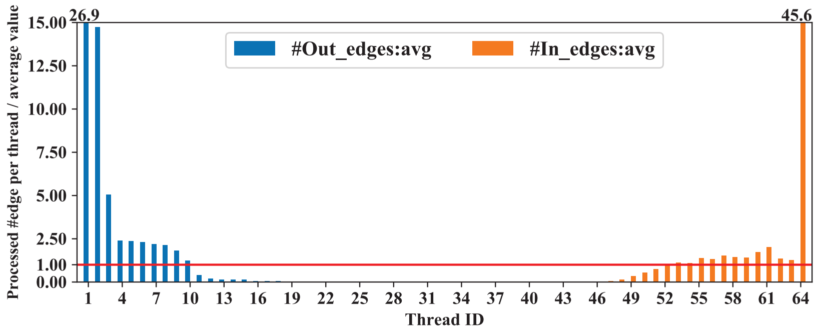
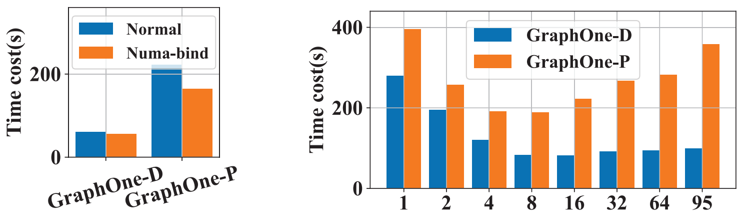


Fig. 4. Imbalanced work division in GraphOne.

sequential writes in GraphOne-P is slightly slower than that in GraphOne-D, due to the hardware bandwidth differences. The main performance bottleneck of GraphOne-P is the graph archiving process, which involves many small data updates (4-byte vertex ID of neighbor information) to the adjacency lists stored in PMEM. Each 4-byte random write may trigger an XPLine (256-byte) read-modify-write to the 3D-Xpoint media according to the Optane's access process, which leads to the severe *PMEM read-write amplification problem*. We also measure the amount of data read from and written to the PMEM during the execution of GraphOne-P, using the Intel PCM tool [62], and show the result in Figure 3(b). We can see that GraphOne-P causes 9.96× read amplification and 8.56× write amplification during the archiving process, which results in high PMEM access costs.

Imbalanced work division for PMEM accesses. As we mentioned in Section 2.1, during the time-consuming graph archiving process, we can divide the batched edges for multi-thread processing. However, the work division strategy used in GraphOne would cause severe workload imbalance among different threads, especially in highly parallel scenarios. We experimentally measure the ratio of the number of edges processed by each thread to the average value when using 64 threads to ingest the Yahoo Web [92] dataset, and show the results in Figure 4. We can see that the work division is highly skewed for both out-graphs and in-graphs. For out-graphs, threads with smaller vertex IDs process more edges than the average. For example, thread 0 processes 21.77× the average number of out-edges, while thread 20 processes only 1% of the average number, and the rest 43 threads are even always idle when processing the out-graphs. For in-graphs, threads with larger vertex IDs process more edges than the average. For example, thread 63 processes 45.65× the average number of in-edges, while the first 64 threads are always idle. In general, the default way of dividing the edge list makes the distribution of the number of edges processed by each



(a) NUMA (b) Number of threads

Fig. 5. Impact of NUMA and number of threads.

thread very uneven. The high cost of accessing graph data stored in PMEM further exacerbates the impact of this imbalanced work division, and greatly reduces the graph processing efficiency.

Costly remote PMEM accesses across NUMA nodes. Moreover, a large portion of these PMEM accesses are cross-NUMA remote PMEM accesses, which further exacerbates the problem of high graph data access cost. We also evaluate the NUMA impact of GraphOne-D and GraphOne-P by comparing the ingestion time of Friendster for normal execution and binding only one NUMA node, respectively, and show the results in Figure 5(a). We can see that GraphOne-P has much larger NUMA effects than GraphOne-D. Furthermore, we conducted tests to evaluate the impact of the number of archiving threads on the performance of GraphOne-D and GraphOne-P. The outcomes, presented in Figure 5(b), indicate that GraphOne-P demonstrates improved performance as the number of archiving threads increases from one to eight, benefiting from parallel processing capabilities. However, a notable decline in performance is observed when the number of archiving threads exceeds eight. This decline can be attributed to PMEM's limited performance in multi-threaded accesses, particularly when dealing with cross-NUMA accesses.

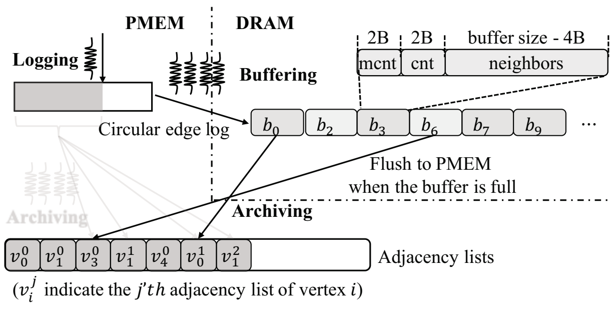
In conclusion, directly moving existing DRAM-based in-memory graph storage systems to PMEM would suffer severe performance drop due to the high PMEM access costs, including high read and write amplification in PMEM, imbalanced work division for PMEM accesses, and costly remote PMEM accesses across NUMA nodes. In the next two sections, we will present our designs to carefully solve the problems, and achieve PMEM friendly graph storing and processing.

3 XPLine-friendly Graph Storage

We target to improve the evolving graph storing performance by reducing PMEM access cost, especially PMEM read-write amplification. A classic solution is to use DRAM buffers to cache and merge data updates to PMEM, but its application to large-scale dynamic graph stores is non-trivial due to the following challenges.

— Evolving graph applications have edge updates with low data locality due to the complex vertex relationships, which makes it hard to design an efficient graph buffering strategy. Moreover, we also need to guarantee the data consistency for the DRAM buffered graph data.
— The buffering strategies usually need high DRAM space for good performance, which may cause the DRAM space pressure, limiting the system scalability. Also, frequent memory allocation and freeing of DRAM buffers may add to the memory management costs.
— Moreover, when ingesting dynamic graph updates that include deletion data, effectively merging the DRAM buffered data and PMEM resident data to conserve space and facilitate fast graph queries poses a significant challenge.

To address these challenges, we introduce an innovative approach called the *XPLine-friendly PMEM access model* based on the XPLine-centric access pattern of PMEM. This model incorporates three key techniques: vertex-centric graph buffering, hierarchical buffer management, and in-place



PMEM
DRAM
Logging
Buffering
2B
2B
buffer size - 4B
mcnt
cnt
neighbors
Circular edge log
b_0 b_2 b_3 b_6 b_7 b_9 ...
Flush to PMEM when the buffer is full
Archiving
v_0^0 v_1^0 v_3^0 v_1^1 v_4^0 v_0^1 v_1^2
Adjacency lists
(v_i^j indicate the j'th adjacency list of vertex i)

Fig. 6. Vertex-centric graph buffering.

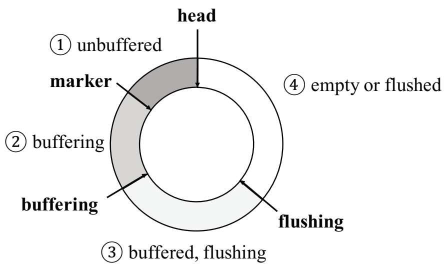


Fig. 7. Consistency guaranteed circular edge log.

vertex block merging. In the following sections, we delve into these techniques in detail and outline how graph data is managed within our proposed model.

3.1 Vertex-centric Graph Buffering

To reduce the read-write amplification problem caused by the *edge-centric adjacency list writes* in existing graph systems, we introduce a *vertex-centric buffering strategy*, to cache some edge updates in DRAM, and lower actual writes to PMEM. In this subsection, we first explain this vertex-centric buffering strategy, and then present the periodical flushing scheme for data consistency. Finally, we analyze the DRAM space cost.

Vertex-centric buffering strategy. As shown in Figure 6, we allocate a temporary adjacency list in DRAM (called vertex buffer in the rest of the article) for each vertex with edge updates, to cache edge updates of the same vertex together. Each vertex buffer has a 4-byte *header* to store the **maximum count** (***mcnt***) and **current count** (***cnt***) of neighbors in this buffer, and the rest space is for storing these neighbors. For example, when the vertex buffer size is 16 bytes, then the buffer can store 3 neighbors of 4-byte vertex ID. When a DRAM vertex buffer is full, we flush all neighbors in it to PMEM by one XPLine access, then clear it for caching subsequent edge updates. With this vertex-centric buffering strategy, we can merge multiple XPLine accesses into one XPLine access, and amortize the PMEM write cost for each edge update. Note that, the buffers can also serve as caches to reduce PMEM reads.

Periodical flushing for consistency guarantees. We implemented a *periodical flushing strategy* for DRAM vertex buffers to ensure fine-grained edge-level consistency in PMEM-resident circular edge logs. As shown in Figure 7, new edge updates are sequentially added to the *head* position in a clockwise order. Once the number of non-buffered edges reaches a threshold, a *buffering phase* begins, buffering these edges to their respective DRAM vertex buffers. When a buffer is full, all neighbors in it are flushed to PMEM. The edges between the *marker* and *buffering* positions are being buffered, while those after the *buffering* position are already buffered. To prevent overwriting buffered edges in the circular edge log, a *flushing phase* is triggered when the number of buffered edges reaches a threshold. All DRAM vertex buffers, including out-neighbors and in-neighbors, are flushed to PMEM for data persistence. A *flushing* pointer marks where edges have already been flushed, allowing new updates to overwrite subsequent data.

To recover lost DRAM vertex buffers after a system crash, we can use the edges between the *marker* and *flushing* positions of the circular edge log. Some edges may have been flushed to PMEM before when their corresponding vertex buffers were full. Therefore, before adding a neighbor to the DRAM vertex buffer, we need to check whether it is already stored in the corresponding PMEM adjacency list to avoid duplicates.

DRAM space requirements for vertex buffers. The performance benefit of the vertex-centric buffering strategy is traded off with the per-vertex buffer size setting. A larger buffer size improves performance by caching more neighbors in DRAM for each vertex, reducing graph ingestion time. However, this comes at the cost of increased DRAM space requirements, limiting system scalability. In our experiments with Yahoo Web [92] graph in Section 5.7, we observed that setting a larger per-vertex buffer size reduced ingestion time but required over 50 GB of DRAM for medium-sized graphs. This setup even made it impossible to handle larger graphs like Kron30 [31] on our 128 GB DRAM server. In the next subsection, we will explore methods to reduce DRAM space requirements while retaining performance benefits.

3.2 Hierarchical Vertex Buffer Managing

In this subsection, we address the issue of DRAM space requirements by introducing an adaptively hierarchical buffer size adjusting scheme based on the power-law vertex degree distribution observed in real-world graphs. We also implement a buddy-like memory pool to reduce the time cost of frequent allocation and freeing of vertex buffers.

Power-law degree distribution of real-world graphs. In real-world graphs, vertex degrees, i.e., the number of neighbors, vary greatly and follow a power-law distribution [11, 29]. For instance, in our evaluated four real-world graphs (refer to Section 5.1), vertices with a degree of 1 or 2 can account for over 40%, and vertices with degrees ranging from 4 to 7 make up around 20%, while only a small portion of vertices have degrees larger than 64. This uneven distribution poses a challenge for allocating buffer sizes, as assigning large buffers to low-degree vertices wastes DRAM space, and allocating small buffers to high-degree vertices hinders the benefits of vertex-centric buffering. To address this issue, we aim at implementing differentiated buffer size settings for vertices based on their degrees. However, predicting the final degree of each vertex in evolving graph situations remains a challenge.

Adaptive hierarchical buffer size adjusting. To address this challenge, we propose an adaptively hierarchical vertex buffering scheme, which adaptively adjusts the vertex buffer sizes according to the changes in vertex degrees, as shown in Figure 8. Initially, a 16-byte buffer (L0) is created for each vertex with edge updates, accommodating up to 3 neighbors. Once the L0 buffer is full, a double-sized (32-byte) L1 buffer is created, and the buffered neighbors are moved there, freeing the L0 buffer. Subsequently, new edge updates are directly written to the L1 buffer, which can hold up to 7 neighbors. This process continues, creating larger buffers as needed until the buffer size reaches the predefined maximum (e.g., 256 bytes). When the largest buffer is full, we flush its neighbors to the corresponding persistent adjacency list stored in PMEM with one XPLine access and then clear the buffer. This adaptive strategy reduces DRAM space requirements for low-degree vertices while preserving performance benefits for high-degree vertices achieved by larger buffers.

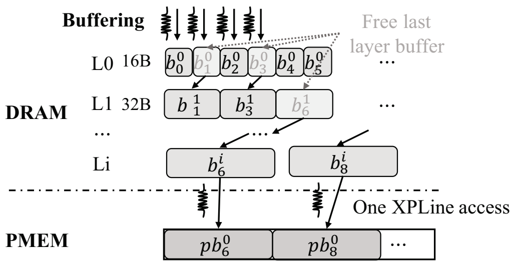


Fig. 8. Adaptive hierarchical buffer size adjusting.

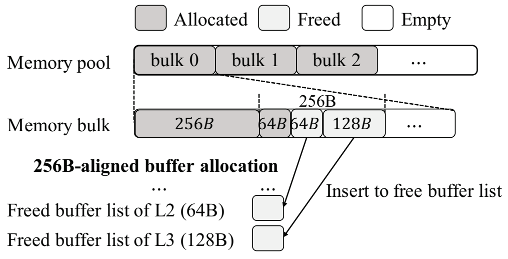

Allocated
Freed
Empty
Memory pool
bulk 0
bulk 1
bulk 2
Memory bulk
256B
64B
64B
128B
256B-aligned buffer allocation
Insert to free buffer list
Freed buffer list of L2 (64B)
Freed buffer list of L3 (128B)

Fig. 9. Memory pool based vertex buffer management.

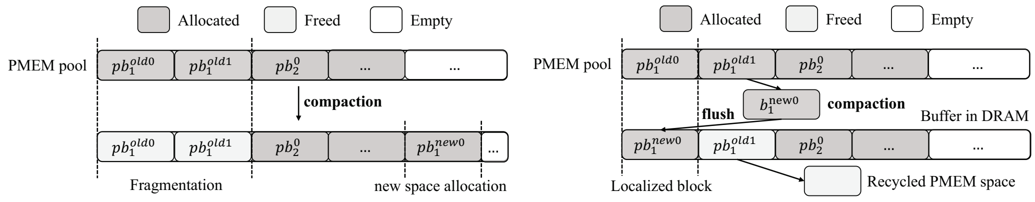
This hierarchical buffer size adjusting strategy incurs extra data movement overhead across layers, but it is acceptable for two reasons. First, the data copy operation does not happen frequently, because we buffer the edge updates in batches (see Section 3.4). If a vertex's edge count reaches a higher level in the current batch, we skip allocating lower layer buffers. Second, the main performance bottleneck lies in PMEM accesses, so the limited DRAM data movement cost can be hidden. The efficiency of this strategy is studied in Section 5.7.

Buddy-liked memory pool management. With the adaptively hierarchical buffer management, we allocate and free small DRAM pieces of varying sizes frequently, resulting in high memory management costs, especially in high-concurrency multi-threading scenarios due to frequent system user mode/kernel mode switches, lock contention, and memory fragmentation. To reduce these costs, we use memory pool management, which pre-allocates a large memory space to handle buffer allocation/freeing by itself, as shown in Figure 9. To avoid access conflicts among threads, We divide the memory pool into smaller memory bulks (default size: 16 MB), and each thread acquires a separate memory bulk to satisfy its buffer allocations. Within each thread's contiguous memory bulk, we employ a buddy-like buffer allocation/free strategy, treating vertex buffers with the same size and adjacent memory space as *buddy buffers*. To ensure scalability, we set a threshold for the memory pool size to limit DRAM usage of all vertex buffers. If the memory pool is close to exhaustion, we flush all vertex buffers to PMEM to reclaim space. We evaluate the impact of memory pool sizes in Section 5.8 to demonstrate performance sensitivity.

3.3 In-place Vertex Block Merging

In dynamic graph scenarios involving evolving graph updates, while the majority of graph updates consist of edge additions, there are also occasional deletions of edges and vertices. These deletions may involve actions such as removing friend relationships or users in social networks.

Tombstone-based deletion strategy. In order to efficiently manage edge and vertex deletions in dynamic graph scenarios, we implement a tombstone-based deletion strategy inspired by the



(a) Naive compacting method (b) Localized block-reused compacting method

Fig. 10. Different neighbors compacting methods.

approach used in GraphOne [44]. The deletion process is structured into three distinct phases. The first phase is the logging phase. When an edge deletion $-e$ occurs, we record a new edge log entry in the circular edge log. To signify the deletion, we set the **most significant bit** (**MSB**) of the source vertex ID to 1. For vertex deletions $-v$, all edges connected to the vertex are logged in the circular edge log and marked as deleted. This methodology ensures that both edge and vertex deletions are effectively managed in evolving graph environments. During the archiving phase, where a batch of edge logs is transferred to their corresponding adjacency lists, we first retrieve all neighbors of the source vertex linked to the deleted edges. This aids in identifying which neighbors need to be removed. In cases where an edge deletion is deemed invalid due to the absence of a preceding edge addition, the deletion is disregarded. Valid edge deletions result in the storage of neighbor indexes in the adjacency list, labeled as new neighbors. Once archiving is complete, we proceed to the final compacting phase, where the actual deletions occur. In this step, we scan the adjacency lists of all vertices, utilizing the stored indexes to eliminate the deleted edges. To enhance the efficiency of the compacting process, we introduce a mechanism to monitor the number of deleted edges associated with each vertex, known as the *deleted degree*. Vertices with a deleted degree of zero are skipped during scanning to expedite the process.

Inefficient PMEM space management for neighbor compacting. In the final compacting phase, GraphOne employs a strategy where new space in DRAM is allocated for compacted adjacency lists based on the disparity between the total degree and deleted degree of a vertex. Subsequently, valid edges are transferred to the new space, and the old space is freed, as depicted in Figure 10(a). While this method functions effectively in DRAM-based systems with manageable allocation and freeing costs, its direct adaptation to PMEM-based systems presents three significant challenges. First, deleted edges may be dispersed across both DRAM and PMEM, necessitating meticulous data migration management between these storage mediums for optimal efficiency. Second, the frequent allocation of PMEM space leads to a substantial degradation in compaction performance compared to DRAM-based systems, as PMEM allocation entails higher overhead. Third, the absence of a system-managed memory reclamation mechanism in PMEM complicates the efficient reclamation of freed space. This results in pronounced external fragmentation issues and wastage of PMEM space. For instance, when applying the naive compacting method to handle a Yahoo Web graph with 6.6 billion inserted edges and a random deletion of 4% of its edges, the PMEM usage for adjacency lists increased by 37%.

Localized block-reused compacting. In response to the challenges outlined earlier, we introduce a localized block-reused compacting technique that provides an efficient solution for storing compacted neighbors in PMEM. As illustrated in Figure 10(b), this method involves allocating a temporary adjacency list block in DRAM to accommodate the compacted neighbors when compacting the neighbors for a vertex v. There are two primary reasons for utilizing DRAM for temporary storage. Firstly, the deletion process involves numerous vertex-level operations, which could result in substantial write amplification in PMEM. Secondly, certain vertices may have all their edges stored in hierarchical buffers that have not yet been flushed to PMEM. For these vertices, retaining the compacted results in DRAM enhances query performance. Subsequent to compaction, the compacted neighbors are transferred to the existing adjacency lists in PMEM, obviating the need for additional PMEM space allocation. Should any of the old adjacency lists become empty, they are repurposed into a list for future use, further reducing PMEM space wastage. It is essential to acknowledge that while there may be some unused space in the old adjacency lists, it is reserved for future edge ingestions and will not go to waste. Although this approach introduces some additional data movement overhead, it is minimal in comparison to the advantages of decreased PMEM write amplification and allocation overhead. Notably, this method does not introduce extra DRAM space overhead, as the temporary space is promptly freed post-compaction. In our evaluation, as outlined in Section 5.3, this approach shows substantial improvements in deletion performance by up to 5.42×, while also reducing PMEM space usage by up to 88% across various graph datasets.

3.4 Data Management Phases

For a better understanding, we describe how graph data is managed in DRAM and PMEM, which may go through three phases in XPGraph: logging, buffering and flushing.

Logging phase. A dedicated logging thread is activated when edge updates are received from the client. This thread sequentially writes these updates to the PMEM resident circular edge log. To prevent overwriting non-flushed edge updates, the logging process will temporarily pause if there is insufficient space in the circular edge log. Once the number of non-buffered edge updates in the log surpasses a predetermined threshold, the system is notified, and the buffering phase begins.

Buffering phase. Multiple buffering threads are utilized to transfer a batch of non-buffered edges from the edge log to the DRAM vertex buffers. To achieve load balancing and maintain data ordering integrity, we employ the *edge sharding based approach* used in GraphOne [44]. This approach involves dividing the batched edges into multiple temporary ranged edge lists based on the ranges of their source vertex IDs. Each of these ranged edge lists can then be buffered in parallel without the need for atomic instructions. To address load imbalance issues, a larger number of ranged edge lists is created compared to the available threads. Different numbers of ranged edge lists are assigned to each buffering thread, ensuring that each thread handles a roughly equal number of edges. We store these temporary ranged edge lists in DRAM for better performance.

Flushing phase. During the buffering phase, when a vertex's highest layer buffer reaches its capacity, the buffering thread enters a brief flushing phase for that vertex, transferring the neighbors stored in the DRAM buffer to the PMEM adjacency list. At the end of each buffering phase, if the number of non-flushed edges in the edge log surpasses a predetermined threshold, indicating that these edges may soon be overwritten, the system is notified to initiate a flushing phase for all vertices. Additionally, if the vertex buffer pool is close to reaching its maximum capacity, the system is also notified to start a flushing phase for all vertices. This action empties the vertex buffer pool, preparing it for the next buffering phases.

4 Scalable Graph Processing

Expanding on the XPLine-friendly graph storage model, we have developed a scalable graph processing framework that integrates two essential techniques: a multi-thread friendly work division strategy and a NUMA-friendly graph accessing approach. Additionally, we have designed user-friendly graph data access APIs and built several prototype systems for efficiently processing large-scale evolving graphs across different memory systems. Our goal is to provide a comprehensive solution that maximizes performance and usability for graph processing tasks.

4.1 Multi-thread Friendly Work Dividing

We employ multi-threading to expedite edge conversion during the archiving phase. A well-designed workload division strategy is crucial for the efficiency of multi-threading. In this subsection, we present a multi-thread friendly work dividing strategy that adapts to varying graph scales and helps improve workload balance across threads.

Unbalanced workload in existing static work division strategy. During the archiving phase, logged edges are distributed among multiple threads for conversion into an adjacency list format. A common approach is to assign edges to threads based on their source vertex ID. By grouping edges with the same source vertex ID together, the efficiency of adjacency list conversion is improved, and thread contention is reduced. For example, GraphOne divides the total edge data into 256 fixed ranges. However, this static strategy has limitations when applied to larger-scale real-world graphs. Firstly, the static work division strategy often results in an uneven distribution of workload, particularly in scenarios with high concurrency involving a large number of threads (e.g., 96 threads) archiving simultaneously. The imbalance is exacerbated by the power-law distribution of vertex degrees in real-world graphs, leading to uneven distribution of edges across the 256 fixed ranges. Secondly, this strategy tends to allocate more edges to the main execution thread than the average, causing most ranges to be assigned to previously allocated threads. This creates an unbalanced workload distribution among the threads. As shown in Figure 4 and discussed in Section 2.3, when applying the existing static work division strategy to a large real-world graph dataset such as Yahoo Web [92], Thread 0 processes approximately 21.77× more edges than the average, while many threads remain underutilized.

Adaptive work division strategy. In order to overcome the challenges outlined earlier, we propose an adaptive multi-thread-friendly work division strategy. Firstly, we dynamically determine the number of divided ranges based on the batch's edge count. We set the number of ranges to $2^{f(E)-b}$, where E represents the number of edges in the batch, $f(E)$ is a function of the edge count E, and b is a tunable bias parameter. This adaptive approach enables us to adjust the range count according to the size of the processing edges, facilitating more precise workload balancing across multiple archiving threads. Secondly, during the archiving phase, we monitor the utilization of each thread. Tasks are assigned first to idle threads with below-average utilization, promoting improved workload balance. Additionally, for each thread, if the sum of the current range's edges and the edges it is already processing exceeds 95% of the average edge count, we reassign the current range to another thread. This strategy ensures efficient utilization of threads with lower workloads and achieves a more balanced distribution of tasks.

Our strategy hinges on two essential settings: function $f(E)$ and parameter b. The function $f(E)$ is adaptive and correlates with the edge count E. It's imperative to devise a well-balanced growth trend for $f(E)$ to avoid uneven division with too few ranges or excessive thread management overhead with too many ranges. Through testing, we have determined that a logarithmic relationship is a suitable choice. Consequently, we set $f(E)$ to the logarithm of the edge count, denoted as $f(E) = \log_2(E)$. Additionally, the parameter b works in conjunction with $f(E)$ to enhance the robustness across diverse workloads and hardware configurations. For instance, when processing a graph with a uniform edge distribution and a limited number of threads, increasing b can reduce the number of ranges, thereby minimizing the need for frequent thread switching. We conducted extensive experiments on a dual-socket system with 96 threads, varying the value of b to observe performance differences. Our results indicate that a default setting of $b = 4$ consistently delivers optimal performance under these conditions. For example, when processing 2^{16} edges with $b = 4$, our strategy divides the edges into 2^{12} ranges. It is important to note that both $f(E)$ and b are adjustable, and users can fine-tune them according to their specific requirements.

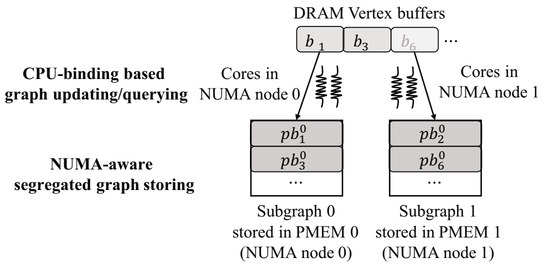

DRAM Vertex buffers
b_1 b_3 b_6 ...
CPU-binding based graph updating/querying
Cores in NUMA node 0
Cores in NUMA node 1
pb_1^0
pb_3^0
pb_2^0
pb_6^0
NUMA-aware segregated graph storing
Subgraph 0 stored in PMEM 0 (NUMA node 0)
Subgraph 1 stored in PMEM 1 (NUMA node 1)

Fig. 11. NUMA-friendly graph accessing.

4.2 NUMA-friendly Graph Accessing

As discussed in Section 2.3, NUMA effects have a significant impact on PMEM compared to DRAM, resulting in a degradation of remote data access performance for graph updates and queries [93]. To mitigate this issue, we propose a novel NUMA-friendly graph accessing strategy, as illustrated in Figure 11. This strategy involves dividing the graph data and storing different parts on different NUMA nodes. Subsequently, the graph updating and querying threads are bound to the CPU cores of the corresponding NUMA nodes.

NUMA-aware segregated graph storing. We introduce a NUMA-aware segregated graph storing approach, where the graph data is partitioned into P parts for a P-socket system. Each part of the graph data is stored in the PMEM(s) of a separate NUMA node. For instance, in a two-socket system, the out-graph data is stored in PMEM 0 of NUMA node 0, while the in-graph data is stored in PMEM 1 of NUMA node 1. This technique can be extended to more general situations, such as undirected graph formats or systems with multiple NUMA nodes, by employing sub-graph based implementations. In the sub-graph-based method, the entire graph is segmented into P sub-graphs using established graph partitioning techniques. Each sub-graph contains the neighbor information for a subset of all vertices. We adopt a widely-used hash-based graph partitioning strategy, where each vertex v is assigned to sub-graph $v\%P$ to ensure a balanced distribution of vertices and edges in each sub-graph. Subsequently, each sub-graph is allocated to the PMEM(s) of a distinct NUMA node. By default, we employ the sub-graph-based implementations, unless otherwise specified. A detailed comparison between the two implementation approaches is provided in Section 5.7.

CPU-binding based graph updating. During the graph buffering and flushing phases, we implement CPU-binding to optimize graph updates for the graph data stored in the PMEM(s) of NUMA node p. This optimization is achieved by assigning the buffering threads to the CPU cores of NUMA node p using the Linux pthread function *pthread_setaffinity_np()* [64]. For example, in the scenario of segregated storing of out/in-graph data, when flushing the adjacency lists of vertices' out-neighbors, the threads are bound to the cores of NUMA node 0. Likewise, when flushing the adjacency lists of vertices' in-neighbors, the threads are bound to the cores of NUMA node 1. In the case of sub-graph segregated storing, when flushing the adjacency lists of sub-graph p, the threads are bound to the cores of NUMA node p. This approach guarantees that remote PMEM writes between NUMA nodes are completely eliminated, as the flushing threads are confined to the local cores, ensuring efficient data processing and minimizing latency.

CPU-binding based graph querying. To prevent remote PMEM reads across NUMA nodes during graph queries, we can bind the querying thread to the CPU cores of the corresponding NUMA node when accessing the adjacency lists of a vertex. However, adopting a per-vertex CPU-binding strategy can result in frequent thread migrations, which incur a high cost. In our experiments, we observed that the cost of thread migrations can be more than ten times that of

Table 1. Graph View APIs

Graph Updating Interfaces	
status	add_edge(src, dst)
count	add_edges(buf, size)
count	buffer_edges(buf, size)
status	del_edge(src, dst)
status	add_vertex(vid)
status	del_vertex(vid)
Graph Querying Interfaces	
count	get_nebrs_{out/in}(vid)
count	get_nebrs_log_{out/in}(vid)
count	get_nebrs_buf_{out/in}(vid)
count	get_nebrs_flush_{out/in}(vid)
count	get_logged_edges()
Graph Arranging Interfaces	
status	buffer_all_edges()
status	flush_all_vbufs()
status	compact_adjs(vid)
status	compact_all_adjs(vid)

remote PMEM access. To mitigate the high cost of frequent thread migrations and avoid remote PMEM reads across NUMA nodes during graph queries, we propose a strategy where vertex queries are classified based on their associated NUMA parts at the beginning of each computing iteration. Subsequently, the querying threads are bound to the corresponding CPU cores before executing the computations. By doing so, we can minimize thread migrations and eliminate remote PMEM reads across NUMA nodes during graph queries.

4.3 Graph View Interfaces and Prototypes

Graph view interfaces. We also provide user-friendly data access APIs. Table 1 outlines the commonly used graph view interfaces. The primary graph updating interface is adding a new edge *add_edge(src, dst)*. We offer several variants to handle batches of edges or delete an existing edge. We also offer interfaces for adding or deleting a vertex. For graph querying interfaces, we offer the basic *get_nebrs_{out/in}(vid)* function to query all out/in neighbors of a vertex. Additionally, we implement various variants of these functions to query out/in neighbors from each data structure. Furthermore, we provide graph arranging interfaces to facilitate the transformation of the graph data status between the three data management phases described earlier. We also implement the *compact_adjs(vid)* function to merge a vertex's adjacency lists stored in both DRAM and PMEM into a single large adjacency list for more efficient graph queries.

Prototype systems. We have implemented all the designs and optimizations described above in approximately 10,500 lines of C++ code encapsulated into a *libxpgraph* library. Using this library, we have developed a basic prototype system called XPGraph, which is a graph storage framework designed for DRAM-PMEM hybrid memory systems. XPGraph enables efficient large-scale evolving graph processing while ensuring fine-grained data consistency for each edge update.

Considering the increasing availability and affordability of battery-backed DRAM [23, 41], we have also implemented a variant of XPGraph called XPGraph-B to cater to these battery-backed systems. In XPGraph-B, we modify the design of the circular edge log to allow the logging process to overwrite the buffered edges. This modification is possible because the DRAM vertex buffers are also included in the power-failure protection domain in battery-backed systems.

Additionally, we have developed another variant system called XPGraph-D to accommodate DRAM-only systems, where all data structures are stored in DRAM, thus supporting situations with enough DRAM space and no crash consistent requirement. In XPGraph-D, the per-vertex buffer size is fixed at 64 bytes to minimize data movement overhead.

To broaden the applicability and scalability of our techniques, we have extended them to **Solid State Drives** (**SSDs**) and developed a variant system named XPGraph-S. XPGraph-S utilizes memory-mapped file I/O to store the circular edge log and persistent adjacency list on SSDs, while retaining vertex buffers in memory. To address the performance challenges typically associated with SSDs, XPGraph-S dynamically adjusts the maximum buffer size for high-degree vertices. This adjustment allows multiple data updates to be consolidated into a single write operation to NAND flash chips, minimizing write amplification on the SSDs. Additionally, XPGraph-S can adapt to various hardware access granularities by dynamically adjusting the hierarchical buffer structure. Moreover, XPGraph-S incorporates a workload-aware vertex merging strategy that considers the specific characteristics of modern SSDs. The performance gap between random and sequential access on modern NVMe SSDs has been significantly reduced. However, the impact of I/O concurrency differs between the two access patterns. As I/O concurrency decreases, the performance of random access deteriorates more rapidly than sequential access. We conducted tests with varying I/O concurrency. When the I/O concurrency dropped to four threads, the performance of sequential reads decreased to 90% of the full-load rate, while random access dropped to only 14%. Therefore, XPGraph-S can intelligently adjust its strategy based on I/O concurrency. During periods of low I/O concurrency, it consolidates all adjacency lists of the queried vertex into a single large list to enhance query performance by reducing the number of I/O operations. As I/O concurrency increases, XPGraph-S proactively avoids merging adjacency lists, instead leveraging high-performance random reads to minimize additional memory copy overhead during ingestion. By extending our techniques to SSDs, XPGraph-S offers improved generality and scalability, enabling efficient processing of large-scale evolving graphs across a broader spectrum of storage systems.

The implementation of XPGraph, along with its variant systems XPGraph-B, XPGraph-D, and XPGraph-S, provides a flexible and efficient solution for processing large-scale evolving graphs across diverse system settings. Furthermore, XPGraph has the potential for future expansion to support emerging storage technologies, such as CXL-based Memory-Semantic SSDs. It is important to highlight that transitioning between the three XPGraph variants can be seamlessly achieved by adjusting specific parameters, making it adaptable to varying workloads and hardware environments.

5 Evaluation

5.1 Experiment Settings

Test bed. We conducted experiments on a server equipped with two 2.10GHz Intel(R) Xeon(R) Gold 5318Y processors, each featuring 24 physical cores with hyper-threading enabled, resulting in a total of 96 logical cores. In terms of memory, the server is equipped with 8 × 16 GB (128 GB) DRAM and 8 × 128 GB (1 TB) Intel Optane Persistent Memory 200 Series. Additionally, the server features a 3.84 TB Intel NVMe SSD for storage purposes. The operating system utilized is Ubuntu 20.04.6 LTS, running on the Linux kernel version 5.4.0.

Comparison systems. As a benchmark for overall performance evaluation, we conducted a comparative analysis using two single-machine graph storage systems, DGAP [37] and GraphOne

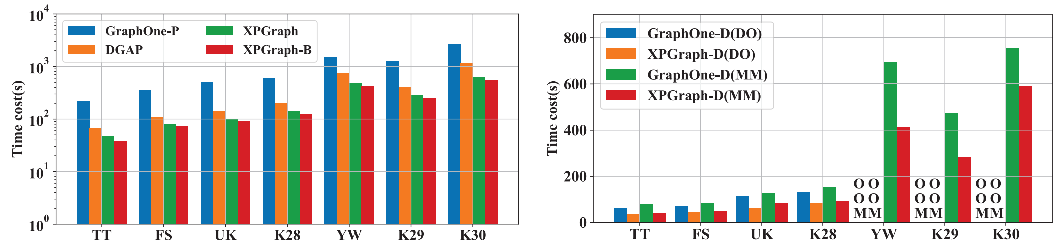
Table 2. Statistics of Datasets

Dataset	$\|V\|$	$\|E\|$	Bin Size	CSR Size
Twitter (TT)	61.6 M	1.5 B	12 GB	12.4 GB
Friendster (FS)	68.3 M	2.6 B	20.8 GB	21.4 GB
UKdomain (UK)	101.7 M	3.1 B	24.8 GB	26.4 GB
YahooWeb (YW)	1.4 B	6.6 B	52.8 GB	75.2 GB
Kron28 (K28)	256 M	4 B	32 GB	36 GB
Kron29 (K29)	512 M	8 B	64 GB	72 GB
Kron30 (K30)	1 B	16 B	128 GB	144 GB
Kron31 (K31)	2 B	64 B	512 GB	576 GB

[44]. DGAP is a state-of-the-art dynamic graph framework for persistent memory, which utilizes a mutable CSR graph structure to deliver high performance. GraphOne is a leading single-machine in-memory graph storage system, and we evaluated on four variants: (1) **GraphOne-D**: The original GraphOne that stores all data on DRAM. (2) **GraphOne-P**: GraphOne on PMEM that the edge log and adjacency lists are stored on PMEM using the default Ext4-DAX file system and the *mmap*-based approach. The meta-data (such as vertex indexes and snapshot information) and intermediate data (such as temporary edge lists) are stored on DRAM. (3) **GraphOne-N**: GraphOne that keeps the meta-data and intermediate data on DRAM, and stores the adjacency lists on PMEM using the NOVA [89] PMEM file system and the *file-I/O*-based approach. To achieve the best performance of NOVA, we mounted NOVA in the *relaxed mode*, which relaxes atomicity constraints on file data and metadata. It is worth mentioning that we also performed the *file-I/O*-based GraphOne on the default Ext4-DAX file system and observed a time cost approximately four times higher than NOVA, which aligns with the results reported in [90]. (4) **GraphOne-S**: GraphOne on SSD that the edge log and adjacency lists are stored on SSD using *mmap* on Ext4 file system. The meta-data and intermediate data remain stored on DRAM.

Graph datasets. We utilized a combination of real-world and synthetic graph datasets for evaluation. The real-world graphs include Twitter [75], Friendster [27], UKdomain [76], and Yahoo Web [92]. Additionally, we incorporated four synthetic Kronecker graphs generated using the graph500 generator [31]. They are all direct graphs, and are widely used in evaluations of graph systems. Table 2 presents detailed information about the graphs. Bin Size indicates the size of the dataset stored in binary edge list format. CSR Size represents the size of storing graphs in the CSR format for both in-graphs and out-graphs. It is important to note that CSR is the most space-efficient storage format for static graphs. However, for dynamic graphs, more memory space is typically required to store evolving adjacency lists for all vertices. For instance, after ingesting the Friendster dataset (which occupies only 21.4 GB in CSR format), GraphOne consumes more than 40 GB of memory solely for storing the adjacency lists. Additionally, it requires an additional 23 GB of memory for storing metadata and intermediate data. As a result, larger graphs like Yahoo Web, Kron29, and Kron30 are not feasible to process using DRAM-based systems (GraphOne-D and XPGraph-D) as their memory demands surpass the 128 GB DRAM capacity of our testing environment.

Evaluation metrics. We assess XPGraph and its comparison systems based on the following four criteria: (1) **Graph ingestion performance** that measures the time required and the amount of PMEM read/write operations involved in ingesting graphs. (2) **Graph deletion performance** that evaluates the time required for deletion operations and the space cost for the adjacency list storage. (3) **Graph query performance** that evaluates the performance of simple one-hop



(a) non-volatile systems (shown in log scale)

(b) volatile systems

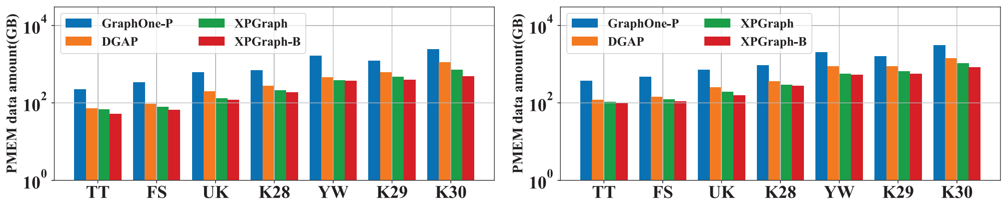
Fig. 12. Graph ingestion time cost for different systems. Optane is set in application-direct mode for non-volatile systems, and for volatile systems, DO indicates running on a DRAM-only system, and MM indicates running on a PMEM-based system with Optane set in memory mode.

neighbor query (which accesses the neighbors of 2^{24} randomly selected vertices with non-zero degrees), as well as three common graph computing algorithms BFS (which traverses the connected sub-graphs of three random roots), PageRank (which runs for ten iterations), and CC (which identifies the connected components in the graphs). (4) **Graph recovery performance** that assesses the efficiency of graph recovery operations. Each experiment was conducted ten times, and the average completion time was calculated to provide a more reliable representation of the time cost. To ensure a fair comparison among different graph systems, all systems were configured according to the G-Bench benchmark [45]. First, we utilized the combined run-time to evaluate overall performance. For system initialization, we adopted the two-parameter method provided by G-Bench to standardize initialization overhead across all systems. Furthermore, we used the binary files to simulate a fast streaming data flow, and use micro-batch to ingest graph data. Additional metrics from the benchmark, such as dedicated analytics and data-access APIs, data generation and shuffling, and graph deletion, were also supported by our system.

5.2 Graph Ingestion Performance

The graph ingestion process imports graph by batching edges from an edge list format. For DGAP, this involves using multiple threads to insert edges into a dynamic edge log while simultaneously updating the PMA tree. For XPGraph and GraphOne, they employ parallel single-thread logging and multi-thread archiving. For simplicity, we refer to XPGraph's buffering and flushing processes as the archiving process throughout the article. In all systems, we use 96 threads (all logical cores on two processors) for ingestion (with archiving threads for XPGraph and GraphOne-P), and we further analyze the impact of the number of archiving threads in Section 5.8. We set the archiving threshold (the number of non-flushed edges stored in the circular edge log required to trigger an archive process) to 2^{16}, similar to the configuration used by GraphOne.

Ingestion time cost for non-volatile systems. We evaluated the time required to ingest the first seven graphs using DGAP, GraphOne-P, XPGraph, and XPGraph-B. All of these systems offer fine-grained edge-level consistency guarantees (with XPGraph-B providing consistency using system-level battery support). Figure 12(a) illustrates the results of this comparison. Compared to DGAP, XPGraph consistently delivers a speedup of 1.37× to 1.80× across all graphs. This improvement can be attributed to two main factors. First, DGAP directly writes the incoming edges (4 bytes) to the edge log on PMEM, which suffers from write performance amplification caused by XPLine. In contrast, XPGraph optimizes batched ingestion by employing hierarchical vertex buffer strategy. Second, DGAP stores edges based on the PMA data structure, which requires frequent updates to the metadata to maintain its balance, along with continuous space allocation and data



(a) PMEM read amount. (b) PMEM write amount.

Fig. 13. PMEM read and write data amount for non-volatile systems (shown in log scale).

migration on PMEM. These operations become especially costly in high-concurrency environments due to increased data contention on PMEM. Although XPGraph involves data conversions from edge log to adjacency list, it leverages the multi-thread friendly work division strategy to evenly distribute the conversion workload across multiple threads and avoid data contention. Consequently, the conversion process is significantly faster, delivering superior overall performance.

Compared to GraphOne-P, XPGraph consistently achieves faster results, with a speedup ranging from 3.07× to 4.99× for various graphs. This improvement is attributed to the reduction of PMEM access costs. Additionally, XPGraph-B, with system-level battery support, further enhances performance by up to 1.4× compared to XPGraph, indicating that the optimizations of XPGraph are applicable to battery-backed systems.

We also evaluated the ingestion performance on GraphOne-N, and GraphOne-N consistently exhibits significantly slower performance compared to the other systems. This can be attributed to the *file-I/O-based* implementation in GraphOne. While NOVA optimizes file I/O operations through high-concurrency log management and bypassing the page cache, it still incurs substantial overhead compared to memory-based operations. It includes costs associated with the **virtual file system** (**VFS**), frequent context-switching, metadata management in the physical file system, data synchronization, and log management in NOVA [90]. In our opinion, the *mmap* based implementation, where the user application is responsible for managing its own address space of PMEM, is more suitable for graph processing scenarios due to its lightweight management overhead.

Ingestion time cost for volatile systems. We evaluated the performance on different hardware settings to demonstrate the versatility of our approach. Specifically, we compared the performance of XPGraph and GraphOne on a DRAM-only system and a PMEM-based system with Optane in memory mode. These systems do not require crash consistency. DGAP was excluded from the comparison as it does not support DRAM-only setups. Figure 12(b) presents the results. When running on the DRAM-only setting, both GraphOne-D and XPGraph-D were unable to handle the three larger graphs due to the limited capacity of DRAM, as they store all data in DRAM. In other test scenarios, XPGraph-D consistently outperformed GraphOne-D. The speedup achieved was up to 1.85× for DRAM-only systems and 1.95× for PMEM-based systems with Optane in memory mode.

PMEM read and write data amount. We measured the data amount read from and written to PMEM during the ingestion process using the Intel PCM tool. This allowed us to verify the significant reduction in PMEM access cost achieved by XPGraph. Figure 13 shows the results. In comparison to GraphOne-P, XPGraph significantly reduced the amount of PMEM read data by 61% to 79%, and PMEM write data by 58% to 78% across different graphs. This improvement was primarily attributed to the vertex-centric local batched graph archiving technique and directly contributed to the overall performance improvement. In comparison to DGAP, XPGraph reduced the PMEM read data amount by up to 56% and PMEM write data amount by up to 54%. This

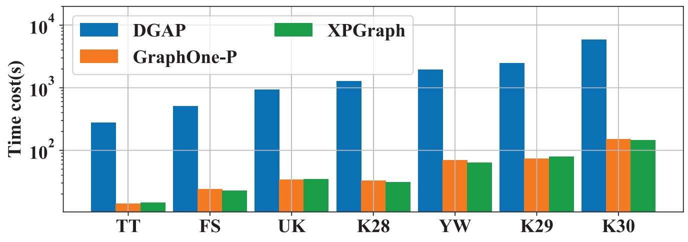


Fig. 14. Preprocess time cost for DGAP, GraphOne-P and XPGraph (shown in log scale).

is because DGAP suffers from write performance amplification caused by XPLine and frequent data migration during rebalance, while XPGraph utilizes hierarchical buffer to minimize the write amplification. Furthermore, XPGraph-B further reduced the PMEM read data amount by up to 47% and PMEM write data amount by up to 25% compared to XPGraph.

Preprocessing cost. Finally, we discuss the preprocessing cost for different systems. For GraphOne, XPGraph, and all their variants, preprocessing involves loading the binary edge list file into memory or PMEM. For DGAP, it involves loading the binary file, inserting the edges into its data structure using an STL vector, and then sorting the edges based on the source vertex ID. Figure 14 illustrates the preprocessing time for the seven graphs. We observed that XPGraph consistently outperformed DGAP, achieving speedups ranging from 19.27× to 40.33× for different graphs. For instance, when ingesting the Kron30 dataset, DGAP required 5824 seconds to insert edges into the vector and sort them, while XPGraph completed the process in just 144 seconds. DGAP's slower performance stems from the high overhead associated with STL vector insertion and sorting operations, which worsens with larger graphs. GraphOne-P exhibits similar preprocessing times to XPGraph, as both systems follow comparable preprocessing steps.

5.3 Graph Deletion Performance

Deletion time comparison. To demonstrate the efficiency of our in-place vertex block merging design on PMEM-based systems, we evaluated the graph deletion performance of XPGraph compared to GraphOne-P. DGAP was excluded from the comparison because the latest version of the DGAP code repository[1] does not support deletion operations. We used the naive compacting method used in GraphOne as a baseline and compared it with our proposed localized block-reused compacting method. For each graph, we ingested all edges, with 4% of them randomly marked for deletion. Once the insertion and deletion data were ingested, we executed the compaction process. We measured the time and space overhead during these two processes across seven graphs. Figure 15(a) presents the time cost of ingestion and compaction. We observed that XPGraph consistently outperforms GraphOne-P, achieving a speedup ranging from 4.02× to 5.42× for different graphs.

PMEM space cost for ingestion and compaction. Additionally, to further analyze the performance gains, we measured the PMEM space consumption during both the ingestion and compaction stages. As shown in Figure 15(b), compared with GraphOne-P, XPGraph significantly reduces the PMEM space usage by 38% to 88% across the two stages. The most substantial space savings are observed during the compaction phase, where XPGraph reduces PMEM space allocation by 22% to 92%. The improved performance and reduced space overhead can be attributed to XPGraph's block-reuse strategy, which minimizes the need for additional PMEM space allocation.

[1]https://github.com/DIR-LAB/DGAP.git

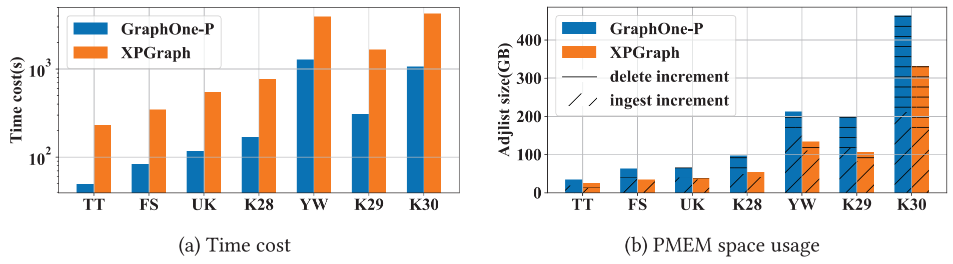


Fig. 15. Graph deletion time cost and PMEM space usage (deletion time is shown in log scale).

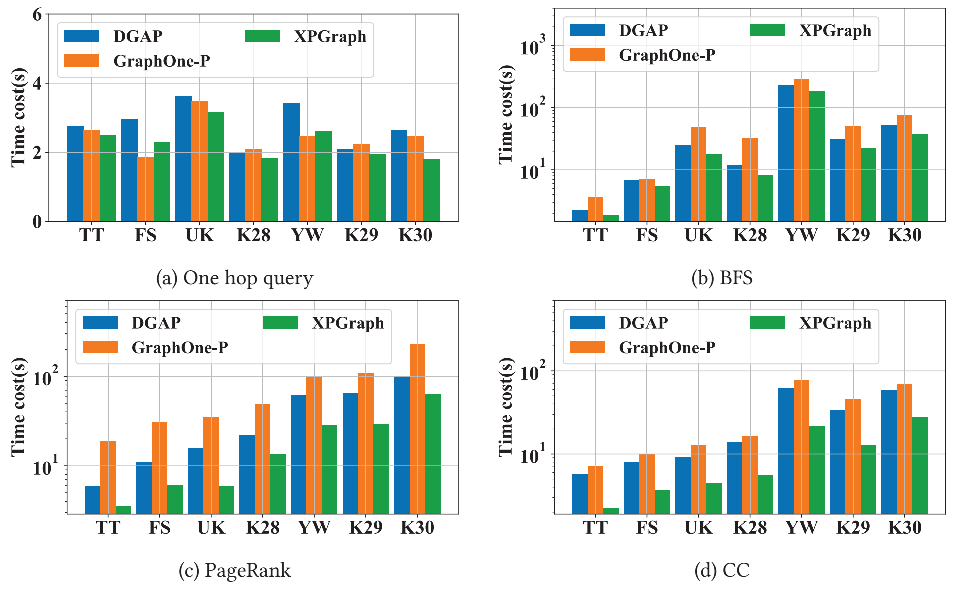


Fig. 16. Graph query performance (BFS, PageRank, and CC are shown in log scale).

In most cases, the adjacency list storage requires less space after deletions, allowing compacted neighbors to fit within the originally allocated space, thereby reducing allocation overhead. Moreover, XPGraph reclaimed and reused the deleted vertex space, further improving the space efficiency.

5.4 Graph Query Performance

To compare the graph query performance, we executed common graph algorithms and aligned the algorithm implementations of XPGraph, GraphOne, and DGAP with a graph processing benchmark suite that standardizes graph processing evaluations [2, 3]. The graph algorithms were executed using all 96 available threads, and the query time costs of GraphOne-P, DGAP, and XPGraph are presented in Figure 16. The sub-figures in the graph depict the time taken to complete four different graph algorithms: one-hop neighbor query, BFS, PageRank, and CC. When considering the one-hop neighbor query, the performance of GraphOne-P, DGAP, and XPGraph varies across different graphs, with comparable time costs in most scenarios and a performance gap limited to 50%. In terms of graph analytic algorithms like BFS, XPGraph consistently outperforms GraphOne-P, achieving speedups of up to 3.96×. Compared to DGAP, XPGraph achieves

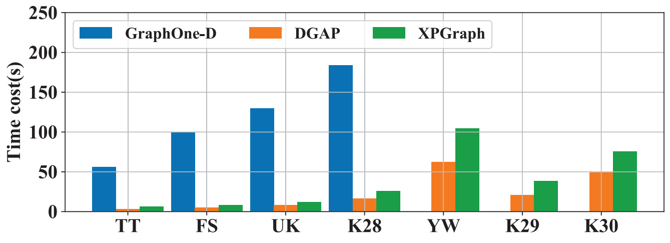


Fig. 17. Graph recovery performance.

speedups ranging from 1.20× to 1.42×. For PageRank and CC algorithms, XPGraph consistently outperforms GraphOne-P with speedups of up to 5.87× and 4.62×, respectively. In these scenarios, XPGraph also outperforms DGAP with speedups of up to 2.92× and 2.68×, respectively. The improved performance of XPGraph can be attributed to two key factors. Firstly, XPGraph caches recent neighbor information in DRAM vertex buffers, reducing the amount of data that needs to be fetched from adjacency lists stored in PMEM. In contrast, GraphOne-P stores all neighbor information in PMEM. Secondly, XPGraph utilizes a NUMA-friendly graph access strategy by evenly distributing PMEM queries to different sockets through sub-graph-based NUMA-aware segregated graph storage. This approach avoids remote PMEM reads across NUMA by using CPU-binding-based graph querying, which enhances the efficiency of PMEM graph data reads. Both GraphOne-P and DGAP do not support NUMA-friendly graph access strategies, resulting in higher PMEM access costs and slower query performance.

5.5 Graph Recovery Performance

PMEM enables data persistence, allowing for quick graph recovery after a power failure. We have implemented a simple recovery scheme in XPGraph that reloads graph data from the persistent adjacency lists stored in PMEM. This scheme also updates the pointer links between multiple adjacency lists of the same vertex, allowing XPGraph to handle new edge updates and graph queries as usual. To evaluate the recovery performance, we compared XPGraph with DGAP and GraphOne. DGAP's recovery process involves loading the data from PMEM using PMDK and rebuilding the data structure. and GraphOne involves rebuilding the data structure by running the archiving process on a bulk of data [44]. We tested GraphOne's recovery performance by measuring the graph archiving time with the archiving threshold set to 2^{27} edges, as recommended in its article for optimal performance. Figure 17 presents the graph recovery time costs for different datasets, including the time to load data from PMEM and recover the adjacency list structure for all vertices. XPGraph consistently outperformed GraphOne in terms of recovery performance, as it does not require rebuilding the data structure for all edges. XPGraph achieved 7.10× to 12.26× higher recovery performance compared to GraphOne for the four relatively small graphs. Even for the larger graphs that GraphOne-D could not handle, XPGraph still achieved reasonable recovery times. For instance, XPGraph only took 75.59 seconds to recover the largest dataset Kron30, whereas GraphOne required 184.08 seconds to recover the much smaller graph Kron28, which had only 1/16th of the edges in Kron30. In contrast, DGAP consistently outperforms XPGraph in recovery performance, achieving speedups ranging from 1.46× to 1.91×. This improvement can be attributed to DGAP's PMA-based data structure, which eliminates the need to update pointer links between multiple adjacency lists for the same vertex during recovery. Additionally, DGAP leverages the PMDK library to accelerate data loading, further enhancing recovery performance. It is important to note that DGAP does suffer from high preprocessing overheads when ingesting new edges, as discussed in Section 5.2.

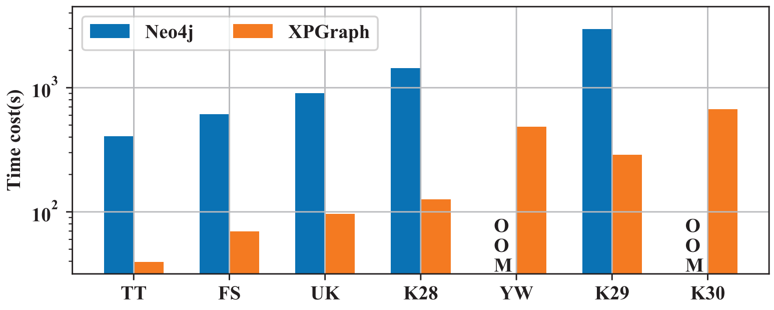


Fig. 18. Graph ingestion performance compared with graph database Neo4j (shown in log scale).

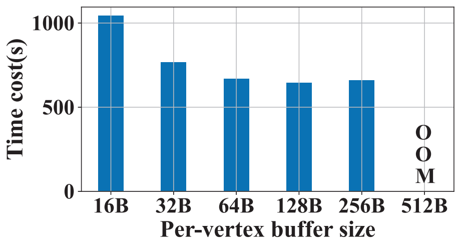
5.6 Comparison with Graph Database

We also compare with the famous graph database Neo4j [58] to assess the improvements in ingestion and query performance. We utilized the community code version 4.4.6 [57] of Neo4j and configured it to use PMEM for graph storage in order to ensure durability and support for large-scale graph datasets. The deployment was on a single server with strong consistency guarantees. Note that both Neo4j and our XPGraph offer fine-grained edge-level consistency guarantees. In terms of ingestion performance, as depicted in Figure 18, XPGraph achieved a remarkable speedup of 8.78× to 11.40× on five smaller graph datasets. However, Neo4j encountered out-of-memory errors and failed to ingest the larger datasets, YahooWeb and Kron30. Regarding query performance, Neo4j struggled to complete a one-hop query benchmark on Friendster even after 1 hour, while XPGraph was able to finish the query in just 2.04 seconds. These improvements primarily stem from XPGraph's efficient PMEM access strategy and NUMA-friendly graph storage design. In contrast, Neo4j suffers from high memory consumption, inefficient PMEM access, and a tradeoff between read/write performance and consistency guarantees for transactional operations. Overall, our system exhibits significant improvements in both ingestion and query performance, making it a more efficient choice for handling graph datasets.

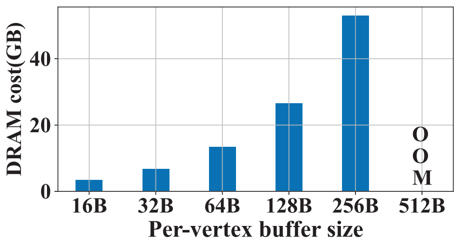
5.7 Discussion of Design Choices

Efficiency of vertex-centric graph buffering. To evaluate the efficiency of the vertex-centric graph buffering strategy described in Section 3.1, we implemented this strategy with a predefined per-vertex buffer size. We conducted experiments using different buffer size settings on the Yahoo Web dataset. Figure 19(a) shows the time cost of ingesting the dynamic graph with different buffer size settings. By buffering some edge updates in DRAM, we observed a significant reduction in the time required for ingestion. Moreover, increasing the buffer size allowed for more neighbors to be cached in DRAM, resulting in fewer total writes to PMEM and further reducing the overall time cost. Figure 19(b) illustrates the DRAM space requirements for the vertex buffers under different buffer size settings. It is important to note that larger buffer sizes require more DRAM space. For instance, when the buffer size is set to 256 bytes, the overall DRAM space demand exceeds 52 GB for the Yahoo Web dataset. Setting the buffer size to 512 bytes even led to an out-of-memory error, limiting scalability for large-scale graph processing. Additionally, the high memory allocation cost for these vertex buffers may slightly degrade performance, resulting in a minor performance drop when increasing the per-vertex buffer size from 128 bytes to 256 bytes.

Efficiency of adaptive hierarchical buffering. We also propose an adaptive hierarchical vertex buffer managing strategy to reduce the DRAM requirement (see Section 3.2). To evaluate its efficiency, we conducted experiments using the same settings as before and present the results in Figure 20. By adopting the hierarchical vertex buffer managing strategy, we achieved the same

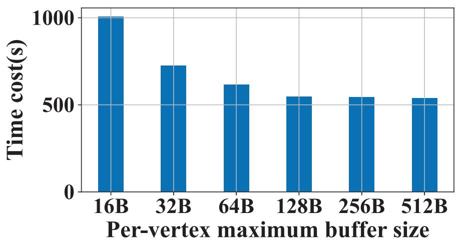


(a) Time cost

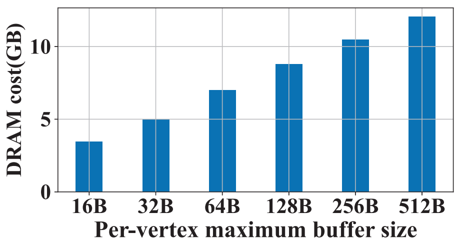


(b) DRAM demand

Fig. 19. Impact of vertex-centric graph buffering with different per-vertex buffer sizes, where OMM indicates the out-of-memory error.



(a) Time cost



(b) DRAM demand

Fig. 20. Efficiency of hierarchical vertex buffer managing with different per-vertex maximum buffer sizes.

performance gains as allocating the maximum buffers for all vertices, while significantly reducing DRAM space requirement. For instance, when a fixed buffer size of 128 bytes was set for all vertices, importing the Yahoo Web dataset took approximately 645.42 seconds and required around 26.54 GB of DRAM space to store all vertex buffers. This was the best performance achieved with a fixed buffer size, as shown in Figure 19. In contrast, using a hierarchical buffer size ranging from 16 to 256 bytes based on vertex degree reduced the ingestion time to 544.72 seconds, due to the lower time overhead of DRAM allocation. Furthermore, the DRAM space requirement decreased to around 10.49 GB, which was less than half of the previous requirement. Additionally, in limited DRAM capacity situations, maximum buffer size can be adjusted to control the DRAM space requirement.

Efficiency of multi-thread friendly work dividing. We conducted two experiments to demonstrate the efficiency of our thread-friendly work division strategy. In both experiments, we used a unified 64 archiving threads to showcase the benefits of our strategy in a multi-threading scenario. First, we compared the ingestion performance of our division strategy with a naive strategy under different numbers of edges of the archiving threshold. We measured the time cost of classifying edges and archiving edges, respectively. The results are shown in Figure 21. The x-axis represents the number of edges of the archiving threshold, and the bars depict the time cost composition for classifying and archiving. Our XPGraph achieved a speedup of 2.23× to 3.49× compared to the naive strategy during the entire ingestion process. The time breakdown reveals that our friendly division strategy outperforms the naive strategy by significantly reducing the time cost of archiving edges. This improvement is due to better workload balance for each thread. Although our strategy incurs slightly higher costs for classifying edges, the tradeoff is acceptable when considering the overall improvement in archiving performance.

In addition, we conducted experiments to evaluate the workload balance achieved by our strategy. We measured the workload of each thread during the archival process for both graph edges

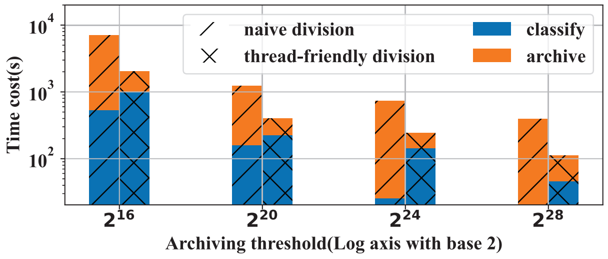


Fig. 21. Graph ingestion time cost using different thread division strategy (shown in log scale).

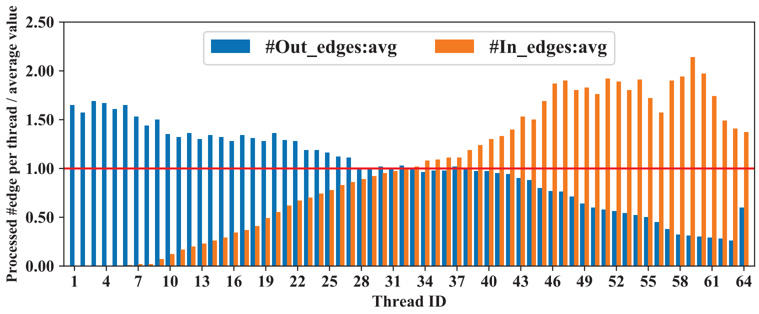
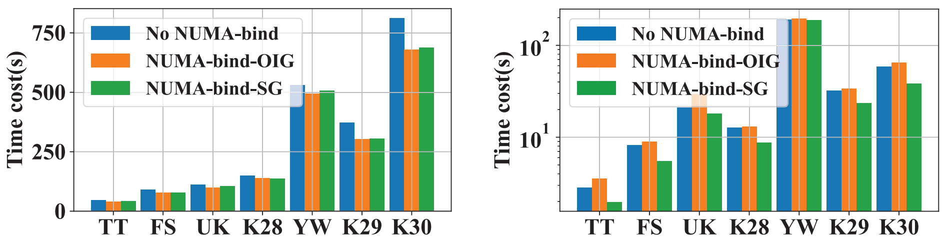


Fig. 22. Per-thread workload distribution using multi-thread friendly work dividing strategy.

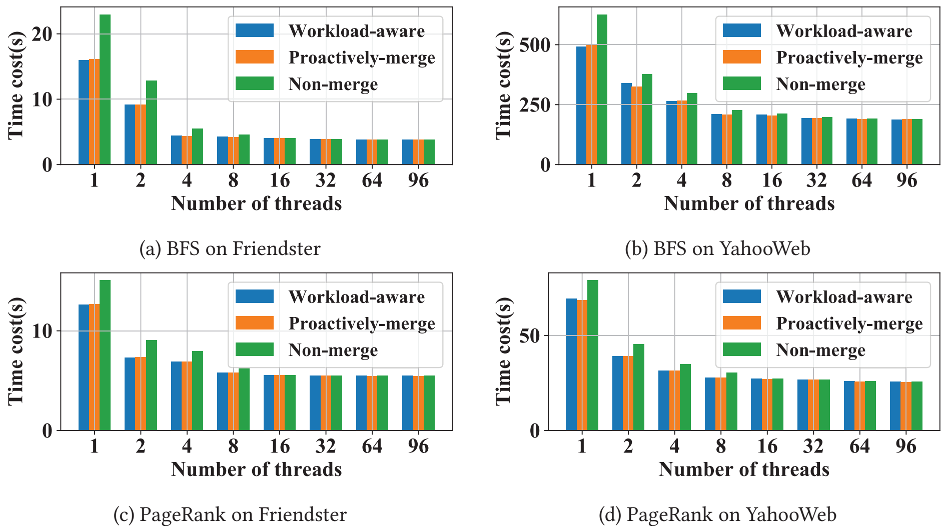


(a) Graph ingestion time (b) BFS time

Fig. 23. Efficiency of NUMA-friendly graph accessing (BFS time is shown in log scale), where NUMA-bind-OIG indicates the out/in-graph-based NUMA-binding implementation, and NUMA-bind-SG indicates the subgraph-based NUMA-binding implementation.

and in graph edges. The results are depicted in Figure 22. The x-axis represents the thread ID ranging from 1 to 64, and the y-axis indicates the ratio of the number of edges processed by each thread to the average value. Compared to the naive strategy workload shown in Figure 4, we can observe that our strategy achieves better workload balance for each thread. This balanced distribution of workload among threads contributes to the overall efficiency and performance of our system.

Efficiency of NUMA-friendly graph accessing. We evaluate the performance improvements obtained by the NUMA-friendly graph accessing method. - Figure 23 illustrates the graph ingestion and BFS computing times under three settings: no NUMA-binding, out/in-graph based NUMA binding, and subgraph based NUMA binding. Enabling the NUMA-friendly graph accessing technique allows us to further enhance graph ingestion performance with speedups ranging from 1.05× to 1.23×. This improvement is particularly significant for larger graphs as it reduces cross-NUMA



(a) BFS on Friendster

(b) BFS on YahooWeb

(c) PageRank on Friendster

(d) PageRank on YahooWeb

Fig. 24. Efficiency of workload-aware vertex merging strategy.

PMEM accesses. Both versions of implementations perform similarly for graph ingestion. However, in terms of graph query performance, the out/in-graph based NUMA binding implementation may result in a performance drop of 3% to 29% due to load imbalance issues. On the other hand, the subgraph based NUMA binding implementation improves BFS performance with speedups of up to 1.54×. This improvement is attributed to the avoidance of remote PMEM accesses across NUMA nodes and the load-balanced sub-graph partitioning across different NUMA nodes.

Efficiency of workload-aware vertex merging strategy. Furthermore, we evaluate the efficiency of the workload-aware vertex merging strategy discussed in Section 4.2 by comparing it with two baseline methods: no merging and proactive merging. Performance of BFS and PageRank algorithms on Friendster and YahooWeb datasets is analyzed (Figure 24), with x-axis denoting the maximum number of available I/O threads and y-axis showing the algorithm's execution time. The results demonstrate that the vertex merging strategies yield substantial performance improvements at concurrency levels below 8, with BFS achieving up to 1.42× on Friendster and 1.25× on YahooWeb, and PageRank up to 1.23× and 1.15×. As concurrency levels increase, the proactive merging strategy's performance aligns with that of the non-merging strategy due to the improved random read performance of SSDs at higher concurrency levels. However, merging vertices introduces extra memory copy overhead, impacting performance at higher concurrency. The proactive merging approach leads to ingestion performance drop of 5.2% on Friendster and 3.7% on YahooWeb, respectively. Therefore, our adaptive merging strategy dynamically selects the optimal approach based on the I/O concurrency level. XPGraph-S is trained to assess the SSD's random read performance at different concurrency levels, enabling it to determine a suitable threshold for proactive merging. In our experimental configuration, this threshold is set to 8. Overall, the adaptive workload-aware vertex merging strategy achieves nearly optimal performance across various I/O concurrency levels.

5.8 Impact of System Configurations

In this subsection, we evaluate the impact of some system configurations, which include vertex buffer pool size, DRAM usage breakdown, different storage devices, the number of archiving threads, and archiving threshold.

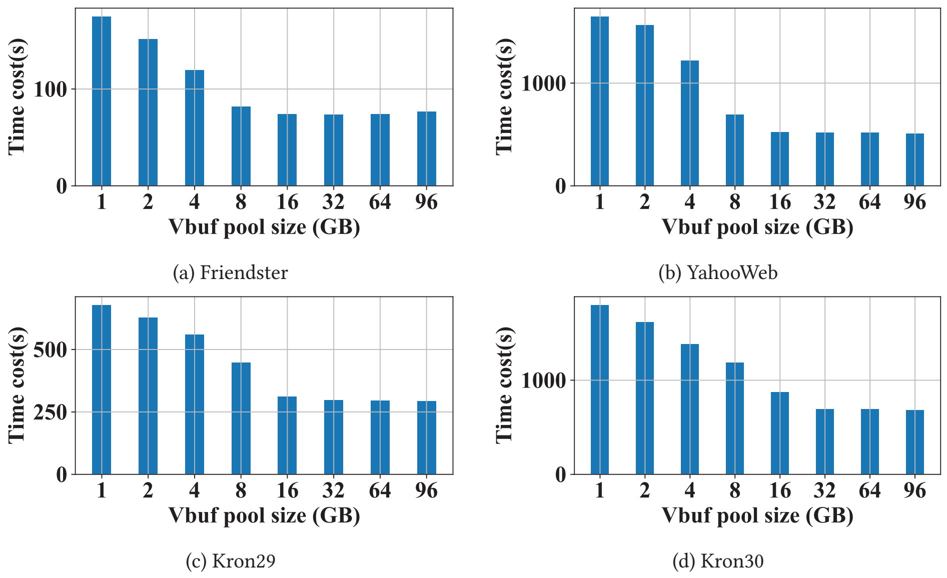


Fig. 25. Impact of vertex buffer memory pool size.

Table 3. Memory Usage of XPGraph (GB)

Dataset	DRAM		PMEM		
	Meta	Vbuf	Input	Elog	Pblk
TT	6.62	5.11	10.94	8.00	13.61
FS	12.11	6.86	19.27	8.00	30.31
UK	16.60	7.08	24.60	8.00	37.19
YW	55.93	10.59	49.57	8.00	116.70
K28	15.72	8.91	32.00	8.00	41.97
K29	27.65	16.89	64.00	8.00	82.67
K30	49.54	28.22	128.00	8.00	165.95

Impact of vertex buffer memory pool size. We conducted experiments to assess the impact of vertex buffer memory pool sizes on the performance of XPGraph. The graph ingestion cost was measured across various memory pool size settings for Friendster, YahooWeb, Kron29, and Kron30, respectively. The results, depicted in Figure 25, indicate that increasing the memory pool size from 1 GB to 16 GB significantly reduces the overall time cost. This improvement is due to the availability of more memory pool space, which allows for caching and merging more PMEM accesses. However, when the memory pool size increases from 32 GB to 96 GB, the time cost remains relatively constant, even for the three largest graphs. This is because 32 GB is typically sufficient to accommodate most vertex buffers for these graphs. Furthermore, the copy-on-write technique used in the Linux system does not actually allocate extra space, so it does not impact performance. Based on these findings, we recommend using a larger memory pool size setting for better performance.

Memory usage breakdown. To analyze the scalability of our design, we provide a breakdown of memory usage during the ingestion process in Table 3. Meta indicates the DRAM usage for storing meta-data and intermediate data, such as vertex indexes, snapshot information, and temporary

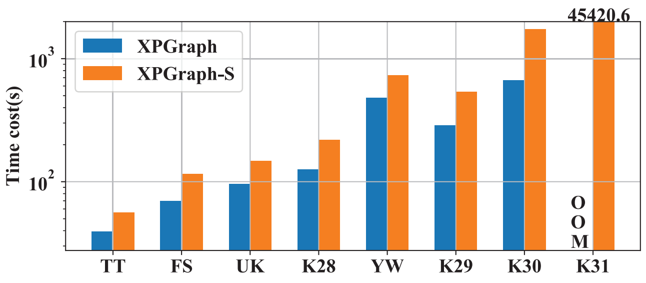


Fig. 26. Graph ingestion performance for different storage devices (shown in log scale), where OOM indicates the out-of-memory error.

edge lists. This portion of DRAM usage is inherited from GraphOne and is similar in XPGraph. It may consume a significant amount of DRAM space and limit the scalability of XPGraph. To further improve scalability, we plan to explore moving this data to PMEM. Vbuf represents the DRAM usage for storing vertex buffers, managed by the vertex buffer memory pool. The memory pool size setting, as shown in Figure 25, can limit this portion of DRAM usage. Input indicates the PMEM usage for storing the input graph data in binary edge list format. Elog represents the PMEM usage for storing the circular edge log, which is set as 8 GB by default. Pblk represents the PMEM usage for storing the persistent adjacency lists, which contain essential graph information in XPGraph. From the table, we observe that XPGraph can efficiently handle large graphs within the capacity of PMEM, while maintaining a limited and tunable DRAM usage. For example, Kron30 with billions of vertices and tens of billions of edges, consumes approximately 80 GB of DRAM space and 300 GB of PMEM space in total. This level of resource consumption can be supported by most medium-sized servers. In servers with larger memory capacity, such as those with terabytes of PMEM per socket, XPGraph can support a significant portion of large-scale graph processing applications.

Impact of different storage devices. In order to accommodate larger graphs that cannot fit in PMEM, we have developed XPGraph-S, a version of our system that utilizes SSD-based storage. To demonstrate the hardware scalability of our work, we conducted a study on the graph ingestion performance using different persistent storage devices. Specifically, we compared the performance of XPGraph and XPGraph-S on a PMEM system and an NVMe-SSD based system, respectively. The results, shown in Figure 26, indicate that XPGraph achieves a speedup of 1.19× to 2.43× compared to XPGraph-S, with the speedup increasing as the graph size grows larger. The superior performance of XPGraph can be attributed to the higher IO bandwidth of PMEM compared to SSD, as well as the lower write amplification achieved through our XPLine granularity buffering strategy. To further showcase the scalability of SSDs when dealing with large-scale graph data, we evaluated XPGraph-S on Kron31, the largest dataset we used. Storing Kron31 in CSR format requires 576 GB of space, and storing all the adjacency lists amounts to over 1.2 TB, surpassing the capacity of our PMEM systems. To overcome this limitation, we divided the graph data into two SSDs and stored all intermediate data in DRAM. As a result, XPGraph-S successfully ingested all the graph data in 12.6 hours. Although the NVMe SSD-based system did cause some performance drops compared to the PMEM-based graph system, it demonstrated enhanced scalability and the ability to handle extremely large-scale graphs.

Impact of the number of archiving threads. We also examined the impact of the number of archiving threads on the ingestion performance of GraphOne-P and XPGraph. The number of archiving threads refers to the number of threads used for converting the graph data from edge

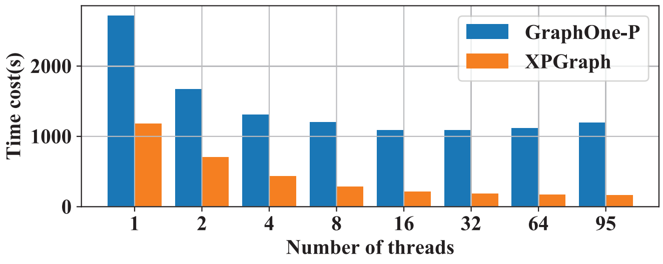


Fig. 27. Impact of number of archiving threads.

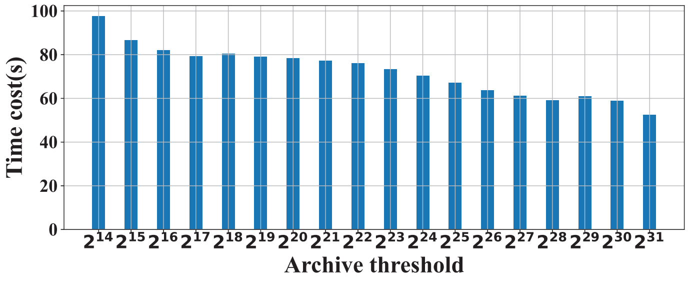


Fig. 28. Impact of archiving threshold.

list format to adjacency list format. Figure 27 presents the ingestion time costs for Yahoo under different settings of the number of archiving threads. Comparing the results with GraphOne-P, we observed that XPGraph consistently outperforms at all thread settings and demonstrates superior scalability for multi-threading. GraphOne-P achieves its best performance under 16 threads, primarily due to the poor PMEM access cost under high concurrent situations. On the other hand, XPGraph exhibits improved ingestion performance as the number of archiving threads increases. This improvement can be attributed to the benefits derived from the multi-thread friendly work dividing and NUMA-friendly graph accessing strategies. These strategies help optimize the utilization of CPU resources and reduce PMEM access latency, resulting in better overall performance. It achieves peak performance when the number of archiving threads is set to 95, which is the maximum number of threads available in our server (with one additional thread used for edge update logging).

Impact of archiving threshold. We further measure the impact of the archiving threshold, i.e., the number of non-flushed edges stored in the circular edge log to trigger an archive process. We show the results in Figure 28, in which the x-axis indicates the number of edges of the archiving threshold, ranging from 2^{14} to 2^{31}, and the y-axis indicates the time cost for completing the graph ingestion for Friendster. We can see that a higher archiving threshold always leads to better performance, which shows the same trend with GraphOne [44]. Specifically, when we set the archiving threshold as 2^{31} edges, XPGraph is able to ingest 49.31 million edges per second, which is 56% faster than the default setting, i.e., 2^{16} edges.

6 Related Work

Large-scale graph processing. Traditional in-memory graph processing systems usually use adjacency list formats, which put edges of the same vertex together, for efficient graph queries [33, 59, 72]. However, their scalability is limited by DRAM capacity. To process large-scale graphs that cannot fit into a single machine's memory, distributed graph systems have been proposed to compute on a cluster of machines [9, 11, 29, 30, 42, 47, 51, 55, 74, 96, 97]. Alternatively, disk-resident single machine graph processing stores graphs on external storage devices [1, 19, 26, 39, 40, 43, 46, 49, 53, 66, 78, 79, 83, 91, 94, 98]. However, these systems are typically designed for static graphs and require efficient graph partitioning, often suffering from performance drops due to communication or disk I/O. Therefore, our XPGraph introduces a PMEM-based graph store that offers both scalability and high-performance graph processing.

Dynamic graph stores. Dynamic graph processing allows for concurrent graph analysis and updates, and various frameworks have been developed in recent years, including STINGER [25], GraphIn [70], EvoGraph [69], Hornet [6], GraphOne [44], and more [18, 32, 38, 56, 65, 71, 84, 85]. These frameworks typically combine the adjacency list and the edge list to balance access locality and update time [6, 25, 44, 69, 70]. Some frameworks employ batching of updates to increase parallelism and throughput [6, 8, 25, 54]. However, these frameworks are designed for DRAM-based dynamic graph stores, leading to efficiency issues when transitioning to a PMEM-based system due to small random accesses during adjacency list updates. Therefore, XPGraph proposes a PMEM-friendly accessing model to support efficient dynamic graph stores for large-scale graphs.

PMEM-based storage optimizations. PMEM has been extensively studied in the past decade [24, 73, 77], even before the release of industrial products, i.e., Intel's Optane DIMM. Due to the different characteristics of PMEM compared to DRAM, such as performance fluctuations influenced by factors like access type, pattern, and size [93], recent works focus on effectively managing data in PMEM for optimal performance. These include PMEM allocators [5, 7, 20, 21, 52, 61, 68, 88], PMEM indexes [10, 12, 14, 17, 34, 36, 50], PMEM-based key-value stores [4, 15, 28, 35, 48, 80, 87], PMEM-based file systems [13, 16, 24, 86, 89, 95], and more. While graph structure data and processing applications differ from these scenarios, it motivates us to design a dedicated PMEM-based graph storage management system XPGraph for efficient large-scale graph processing. Furthermore, recent research [22] highlights the ongoing relevance of PMEM, particularly in light of emerging technologies such as CXL, further motivating us to explore its potential in graph processing.

7 Conclusion

This article proposes XPGraph, an XPLine-friendly PMEM-based graph storage system for high-performance storage of large-scale evolving graphs. XPGraph introduces an XPLine-friendly graph storage model that optimizes the flushing of graph data to PMEM through vertex-centric graph buffering, hierarchical vertex buffer management, and in-place vertex block merging. Additionally, XPGraph develops a scalable graph processing model that enhances PMEM graph accesses through multi-threaded work division and NUMA-friendly graph accessing. We provide data access APIs in the form of a library called *libxpgraph* to facilitate easy use by user applications. XPGraph is implemented along with three variants for different system settings, and experimental results demonstrate significant reductions in PMEM access costs for graph data, as well as improved performance in graph update, query, and recovery tasks.

Acknowledgments

We would like to thank the Transactions on Storage editors and reviewers for all their valuable feedback and suggestions. We also thank Xuechen Zhang, Jian Xu, and all the MICRO reviewers for their help in improving this work.

References

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

[2] Scott Beamer, Krste Asanović, and David Patterson. 2015. The GAP benchmark suite. arXiv:1508.03619. Retrieved from https://arxiv.org/abs/1508.03619

[3] Scott Beamer, Krste Asanovic, and David Patterson. 2015. Locality exists in graph processing: Workload characterization on an ivy bridge server. In *Proceedings of the 2015 IEEE International Symposium on Workload Characterization*. 56–65.

[4] Lawrence Benson, Hendrik Makait, and Tilmann Rabl. 2021. Viper: An efficient hybrid pmem-dram key-value store. *PVLDB* 14, 9 (2021), 1544–1556.

[5] Kumud Bhandari, Dhruva R. Chakrabarti, and Hans-J. Boehm. 2016. Makalu: Fast recoverable allocation of non-volatile memory. *ACM SIGPLAN Notices* 51, 10 (2016), 677–694.

[6] Federico Busato, Oded Green, Nicola Bombieri, and David A. Bader. 2018. Hornet: An efficient data structure for dynamic sparse graphs and matrices on GPUs. In *Proceedings of the IEEE HPEC*. 1–7.

[7] Wentao Cai, Haosen Wen, H. Alan Beadle, Chris Kjellqvist, Mohammad Hedayati, and Michael L Scott. 2020. Understanding and optimizing persistent memory allocation. In *Proceedings of the ACM SIGPLAN ISMM*. 60–73.

[8] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *IEEE TCDE* 38, 4 (2015), 28–38.

[9] Hongzhi Chen, Miao Liu, Yunjian Zhao, Xiao Yan, Da Yan, and James Cheng. 2018. G-Miner: An efficient task-oriented graph mining system. In *Proceedings of the ACM EuroSys*. 1–12.

[10] Qi Chen, Hao Hu, Cai Deng, Dingbang Liu, Shiyi Li, Bo Tang, Ting Yao, and Wen Xia. 2023. EEPH: An efficient extendible perfect hashing for hybrid PMem-DRAM. In *Proceedings of the IEEE ICDE*. IEEE, 1366–1378.

[11] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. 2015. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the ACM EuroSys*. 1–39.

[12] Shimin Chen and Qin Jin. 2015. Persistent B+-Trees in non-nolatile main memory. *PVLDB* 8, 7 (2015), 786–797.

[13] Youmin Chen, Youyou Lu, Pei Chen, and Jiwu Shu. 2018. Efficient and consistent NVMM cache for SSD-based file system. *IEEE TC* 68, 8 (2018), 1147–1158.

[14] Youmin Chen, Youyou Lu, Kedong Fang, Qing Wang, and Jiwu Shu. 2020. UTree: A persistent b+-tree with low tail latency. *PVLDB* 13, 12 (2020), 2634–2648.

[15] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. 2020. Flatstore: An efficient log-structured key-value storage engine for persistent memory. In *Proceedings of the ACM ASPLOS*. 1077–1091.

[16] Youmin Chen, Jiwu Shu, Jiaxin Ou, and Youyou Lu. 2018. HiNFS: A persistent memory file system with both buffering and direct-access. *ACM ToS* 14, 1 (2018), 1–30.

[17] Zhiwen Chen, Wenkui Che, Daokun Hu, Xin He, Jianhua Sun, and Hao Chen. 2023. On the performance intricacies of persistent memory aware storage engines. *IEEE TKDE* 35, 10 (2023), 10365–10382.

[18] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. 2012. Kineograph: Taking the pulse of a fast-changing and connected world. In *Proceedings of the ACM EuroSys*. 85–98.

[19] Disa Mhembere Da Zheng, Randal Burns, Joshua Vogelstein, Carey E Priebe, and Alexander S. Szalay. 2015. Flash-Graph: Processing billion-node graphs on an array of commodity SSDs. In *Proceedings of the USENIX FAST*. 45–58.

[20] Zheng Dang, Shuibing He, Peiyi Hong, Zhenxin Li, Xuechen Zhang, Xian-He Sun, and Gang Chen. 2022. NVAlloc: Rethinking heap metadata management in persistent memory allocators. In *Proceedings of the ACM ASPLOS*. 115–127.

[21] Zheng Dang, Shuibing He, Xuechen Zhang, Peiyi Hong, Zhenxin Li, Xinyu Chen, Haozhe Song, Xian-He Sun, and Gang Chen. 2024. PMAlloc: A holistic approach to improving persistent memory allocation. *ACM TC* 42, 3–4 (2024), 1–52.

[22] Peter Desnoyers, Ian Adams, Tyler Estro, Anshul Gandhi, Geoff Kuenning, Mike Mesnier, Carl Waldspurger, Avani Wildani, and Erez Zadok. 2023. Persistent memory research in the post-optane era. In *Proceedings of the DIMES*. 23–30.

[23] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. No compromises: Distributed transactions with consistency, availability, and performance. In *Proceedings of the ACM SOSP*. 54–70.

[24] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System software for persistent memory. In *Proceedings of the ACM EuroSys*. 1–15.

[25] David Ediger, Rob McColl, Jason Riedy, and David A. Bader. 2012. Stinger: High performance data structure for streaming graphs. In *Proceedings of the IEEE HPEC*. 1–5.

[26] Nima Elyasi, Changho Choi, and Anand Sivasubramaniam. 2019. Large-scale graph processing on emerging storage devices. In *Proceedings of the USENIX FAST*. 309–316.

[27] Friendster Graph. 2025. Friendster. Retrieved from http://konect.cc/networks/friendster/

[28] Xuran Ge, Mingche Lai, Yang Liu, Lizhou Wu, Zhutao Zhuang, Yang Ou, Zhiguang Chen, and Nong Xiao. 2022. SpacKV: A pmem-aware key-value separation store based on LSM-tree. In *Proceedings of the ACM NPC*. Springer, 327–339.

[29] 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 USENIX OSDI*. 17–30.

[30] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. 2014. GraphX: Graph processing in a distributed dataflow framework. In *Proceedings of the USENIX OSDI*. 599–613.

[31] Graph500 Generator. 2025. Graph500. Retrieved from https://graph500.org/.

[32] Oded Green and David A. Bader. 2016. cuSTINGER: Supporting dynamic graph algorithms for GPUs. In *Proceedings of the IEEE HPEC*. 1–6.

[33] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. 2012. Green-Marl: A DSL for easy and efficient graph analysis. *ACM SIGPLAN Notices* 47, 4 (2012), 349–362.

[34] Daokun Hu, Zhiwen Chen, Wenkui Che, Jianhua Sun, and Hao Chen. 2022. Halo: A hybrid PMem-DRAM persistent hash index with fast recovery. In *Proceedings of the ACM SIGMOD*. 1049–1063.

[35] Yihe Huang, Matej Pavlovic, Virendra Marathe, Margo Seltzer, Tim Harris, and Steve Byan. 2018. Closing the performance gap between volatile and persistent key-value stores using cross-referencing logs. In *Proceedings of the USENIX ATC*. 967–979.

[36] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. 2018. Endurable transient inconsistency in byte-addressable persistent b+-tree. In *Proceedings of the USENIX FAST*. 187–200.

[37] Abdullah Al Raqibul Islam and Dong Dai. 2023. DGAP: Efficient dynamic graph analysis on persistent memory. In *Proceedings of the ACM SC*. 1–13.

[38] Anand Padmanabha Iyer, Li Erran Li, Tathagata Das, and Ion Stoica. 2016. Time-evolving graph processing at scale. In *Proceedings of the ACM GRADES*. 1–6.

[39] Myung-Hwan Jang, Jeong-Min Park, Ikhyeon Jo, Duck-Ho Bae, and Sang-Wook Kim. 2023. RealGraph+: A high-performance single-machine-based graph engine that utilizes IO bandwidth effectively. In *Proceedings of the ACM WWW*. 276–279.

[40] Sang-Woo Jun, Andy Wright, Sizhuo Zhang, Shuotao Xu, and Arvind. 2018. GraFBoost: Using accelerated flash storage for external graph analytics. In *Proceedings of the IEEE ISCA*. 411–424.

[41] Rajat Kateja, Anirudh Badam, Sriram Govindan, Bikash Sharma, and Greg Ganger. 2017. Viyojit: Decoupling battery and DRAM capacities for battery-backed DRAM. In *Proceedings of the IEEE ISCA*. 613–626.

[42] Arijit Khan, Gustavo Segovia, and Donald Kossmann. 2018. On smart query routing: for distributed graph querying with decoupled storage. In *Proceedings of the USENIX ATC*. 401–412.

[43] Juno Kim and Steven Swanson. 2022. Blaze: Fast graph processing on fast SSDs. In *Proceedings of the IEEE SC*. IEEE, 1–15.

[44] Pradeep Kumar and H. Howie Huang. 2019. GraphOne: A data store for real-time analytics on evolving graphs. In *Proceedings of the USENIX FAST*. 249–263.

[45] Pradeep Kumar and Sarah Revillar. 2023. G-Bench: Fair benchmarking to support innovations in streaming graph systems. In *Proceedings of the IEEE IPDPSW*. IEEE, 179–188.

[46] Aapo Kyrola, Guy E. Blelloch, and Carlos Guestrin. 2012. Graphchi: Large-scale graph computation on just a PC. In *Proceedings of the USENIX OSDI*. 31–46.

[47] Changji Li, Hongzhi Chen, Shuai Zhang, Yingqian Hu, Chao Chen, Zhenjie Zhang, Meng Li, Xiangchen Li, Dongqing Han, Xiaohui Chen, et al. 2022. ByteGraph: A high-performance distributed graph database in bytedance. *PVLDB* 15, 12 (2022), 3306–3318.

[48] Zhenxin Li, Shuibing He, Zheng Dang, Peiyi Hong, Xuechen Zhang, Rui Wang, and Fei Wu. 2024. CCL-BTree: A crash-consistent locality-aware b+-tree for reducing xpbuffer-induced write amplification in persistent memory. In *Proceedings of the 19th European Conference on Computer Systems*. 441–455.

[49] Hang Liu and H. Howie Huang. 2017. Graphene: Fine-grained io management for graph computing. In *Proceedings of the USENIX FAST*. 285–300.

[50] Jihang Liu, Shimin Chen, and Lujun Wang. 2020. LB+ Trees: Optimizing persistent index performance on 3dxpoint memory. *PVLDB* 13, 7 (2020), 1078–1090.

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

[52] Shaonan Ma, Kang Chen, Shimin Chen, Mengxing Liu, Jianglang Zhu, Hongbo Kang, and Yongwei Wu. 2021. ROART: Range-query optimized persistent ART. In *Proceedings of the USENIX FAST*. 1–16.

[53] 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 ACM EuroSys*. 527–543.

[54] Peter Macko, Virendra J. Marathe, Daniel W. Margo, and Margo I. Seltzer. 2015. Llama: Efficient graph analytics using large multiversioned arrays. In *Proceedings of the IEEE ICDE*. 363–374.

[55] 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 ACM SIGMOD*. 135–146.

[56] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: A timely dataflow system. In *Proceedings of the ACM SOSP*. 439–455.

[57] Neo4j. 2025. Neo4j 4.4.6. Retrieved from https://github.com/neo4j/neo4j/tree/4.4.6

[58] Neo4j Graph Database. 2025. Neo4j Inc. Retrieved from https://neo4j.com/

[59] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A lightweight infrastructure for graph analytics. In *Proceedings of the ACM SOSP*. 456–471.

[60] Optane PMEM. 2025. Intel Optane Persistent Memory. Retrieved from https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html

[61] Ismail Oukid, Daniel Booss, Adrien Lespinasse, Wolfgang Lehner, Thomas Willhalm, and Grégoire Gomes. 2017. Memory management techniques for large-scale persistent-main-memory systems. *PVLDB* 10, 11 (2017), 1166–1177.

[62] PCM Tool. 2025. Intel. Processor Counter Monitor (PCM). Retrieved from https://github.com/opcm/pcm

[63] Vijayan Prabhakaran, Ming Wu, Xuetian Weng, Frank McSherry, Lidong Zhou, and Maya Haradasan. 2012. Managing large graphs on multi-cores with graph awareness. In *Proceedings of the USENIX ATC*. 41–52.

[64] pthread setaffinity np. 2025. pthread_setaffinity_np(3) – Linux manual page. Retrieved from https://man7.org/linux/man-pages/man3/pthread_setaffinity_np.3.html

[65] Hao Qi, Yiyang Wu, Ligang He, Yu Zhang, Kang Luo, Minzhi Cai, Hai Jin, Zhan Zhang, and Jin Zhao. 2024. LSGraph: A locality-centric high-performance streaming graph engine. In *Proceedings of the ACM EuroSys*. 33–49.

[66] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the ACM SOSP*. 472–488.

[67] Samsung Memory-Semantic SSD. [n. d.]. Samsung Memory-Semantic SSD. Retrieved from https://semiconductor.samsung.com/us/news-events/tech-blog/webinar-memory-semantic-ssd/

[68] David Schwalb, Tim Berning, Martin Faust, Markus Dreseler, and Hasso Plattner. 2015. NVM malloc: Memory allocation for NVRAM. In *Proceedings of ADMS*. 61–72.

[69] Dipanjan Sengupta and Shuaiwen Leon Song. 2017. Evograph: On-the-fly efficient mining of evolving graphs on GPU. In *Proceedings of the ACM ISC*. 97–119.

[70] Dipanjan Sengupta, Narayanan Sundaram, Xia Zhu, Theodore L. Willke, Jeffrey Young, Matthew Wolf, and Karsten Schwan. 2016. GraphIn: An online high performance incremental graph processing framework. In *Proceedings of the Euro-Par*. 319–333.

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

[72] Julian Shun and Guy E. Blelloch. 2013. Ligra: A lightweight graph processing framework for shared memory. In *Proceedings of the ACM SIGPLAN*. 135–146.

[73] Ying Tai, Jian Yang, Xiaoming Liu, and Chunyan Xu. 2017. Memnet: A persistent memory network for image restoration. In *Proceedings of the IEEE ICCV*. 4539–4547.

[74] Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga. 2015. Arabesque: A system for distributed graph mining. In *Proceedings of the ACM SOSP*. 425–440.

[75] Twitter Graph. 2025. Twitter. Retrieved from http://an.kaist.ac.kr/traces/WWW2010.html

[76] UKdomain Graph. 2025. UK domain (2007) network dataset – KONECT. Retrieved from http://konect.cc/networks/dimacs10-uk-2007-05

[77] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight persistent memory. *ACM SIGARCH CAN* 39, 1 (2011), 91–104.

[78] Keval Vora. 2019. LUMOS: Dependency-driven disk-based graph processing. In *Proceedings of the USENIX ATC*. 429–442.

[79] Keval Vora, Guoqing (Harry) Xu, and Rajiv Gupta. 2016. Load the edges you need: A generic I/O optimization for disk-based graph processing. In *Proceedings of the USENIX ATC*. 507–522.

[80] Jing Wang, Youyou Lu, Qing Wang, Yuhao Zhang, and Jiwu Shu. 2024. Perseid: A secondary indexing mechanism for LSM-based storage systems. *ACM TS* 20, 2 (2024), 1–28.

[81] Qing Wang, Youyou Lu, Junru Li, and Jiwu Shu. 2021. Nap: A black-box approach to NUMA-aware persistent memory indexes. In *Proceedings of the USENIX OSDI*. 93–111.

[82] Rui Wang, Yongkun Li, Hong Xie, Yinlong Xu, and John C. S. Lui. 2020. GraphWalker: An I/O-efficient and resource-friendly graph analytic system for fast and scalable random walks. In *Proceedings of the USENIX ATC*. 559–571.

[83] Rui Wang, Weixu Zong, Shuibing He, Xinyu Chen, Zhenxin Li, and Zheng Dang. 2024. Efficient large graph processing with chunk-based graph representation model. In *Proceedings of the USENIX ATC*. 1239–1255.

[84] Martin Winter, Daniel Mlakar, Rhaleb Zayer, Hans-Peter Seidel, and Markus Steinberger. 2018. FaimGraph: High performance management of fully-dynamic graphs under tight memory constraints on the GPU. In *Proceedings of the IEEE SC*. 754–766.

[85] Martin Winter, Rhaleb Zayer, and Markus Steinberger. 2017. Autonomous, independent management of dynamic graphs on GPUs. In *Proceedings of the IEEE HPEC*. 1–7.

[86] Hobin Woo, Daegyu Han, Seungjoon Ha, Sam H. Noh, and Beomseok Nam. 2023. On stacking a persistent memory file system on legacy file systems. In *Proceedings of the USENIX FAST*. 281–296.

[87] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. 2017. HiKV: A hybrid index key-value store for DRAM-NVM memory systems. In *Proceedings of the USENIX ATC*. 349–362.

[88] Xiangyu Xiang, Yu Hua, and Hao Xu. 2023. PMA: A persistent memory allocator with high efficiency and crash consistency guarantee. In *Proceedings of the IEEE ICCD*. IEEE, 182–189.

[89] Jian Xu and Steven Swanson. 2016. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *Proceedings of the USENIX FAST*. 323–338.

[90] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. 2017. NOVA-fortis: A fault-tolerant non-volatile main memory file system. In *Proceedings of the ACM SOSP*. 478–496.

[91] Xianghao Xu, Hong Jiang, Fang Wang, Yongli Cheng, and Peng Fang. 2022. Graphsd: A state and dependency aware out-of-core graph processing system. In *Proceedings of the ACM ICPP*. 1–11.

[92] Yahoo Graph. 2025. Yahoo Webscope. Yahoo! AltaVista Web Page Hyperlink Connectivity Graph. Retrieved from http://webscope.sandbox.yahoo.com

[93] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An empirical guide to the behavior and use of scalable persistent memory. In *Proceedings of the USENIX FAST*. 169–182.

[94] Tsun-Yu Yang, Yizou Chen, Yuhong Liang, and Ming-Chang Yang. 2024. Seraph: Towards scalable and efficient fully-external graph computation via on-demand processing. In *Proceedings of the USENIX FAST*. 373–387.

[95] Yiwen Zhang, Jian Zhou, Xinhao Min, Song Ge, Jiguang Wan, Ting Yao, and Daohui Wang. 2022. PetaKV: Building efficient key-value store for file system metadata on persistent memory. *IEEE TPDS* 34, 3 (2022), 843–855.

[96] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A computation-centric distributed graph processing system. In *Proceedings of the USENIX OSDI*. 301–316.

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

[98] Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. GridGraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *Proceedings of the USENIX ATC*. 375–386.

Received 29 September 2024; revised 23 December 2024; accepted 12 January 2025