PIE: Enabling Fast and Scalable Incremental Evolving Graph Analytics on Persistent Memory

Yunmo Zhang
City University of Hong Kong
Hong Kong, China
yunmo.zhang@my.cityu.edu.hk

Jiacheng Huang
City University of Hong Kong
Hong Kong, China
jiacheng.huang@my.cityu.edu.hk

Xizhe Yin
University of California Riverside
Riverside, USA
xyin014@ucr.edu

Junqiao Qiu*
City University of Hong Kong
Hong Kong, China
junqiqiu@cityu.edu.hk

Hong Xu
The Chinese University of Hong Kong
Hong Kong, China
hongxu@cuhk.edu.hk

Chun Jason Xue
MBZUAI
Abu Dhabi, United Arab Emirates
jason.xue@mbzuai.ac.ae

Abstract

Graph processing is crucial for unstructured-data-driven applications in various domains. In recent years, there has been a growing need to perform real-time analytics on large-scale evolving graphs, which involves evaluating a graph query on a sequence of snapshots within a given time window. Some prior studies have explored utilizing persistent memory (PM) technologies, such as non-volatile memory, for efficient evolving graph analytics. However, the latest incremental processing designs fail to fully exploit the PM potential, suffering from severe read and write amplification during update ingestion and query evaluation. In this paper, we develop PIE, a PM-based incremental processing framework for fast and scalable evolving graph analytics. We first observe that leveraging CommonGraph, a recently proposed DRAM-based incremental approach that transforms costly deletions into additions, can significantly improve efficiency for evolving graph analytics in PM, although the direct adaptation introduces significant PM access inefficiencies. To enable PM-friendly incremental processing, PIE introduces a logical graph view abstraction that is detached from the physical storage to avoid extra PM writes, and a chunked neighbor index to reduce extensive PM reads during deletion transformation. Additionally, PIE prioritizes low-cost transformations when executing deletion-free incremental analytics. Furthermore, PIE incorporates two optimizations to balance key trade-offs in the proposed analytics on PM. Experimental results show that PIE significantly outperforms two state-of-the-art PM-based evolving graph systems, DGAP and XPGraph, as well as the direct PM adaptation of CommonGraph, achieving geometric mean speedups of 9.5×, 8.4×, and 10.9× for update ingestion, and 18.1×, 20.4× and 6.4× for graph analytics, respectively.

CCS Concepts

• **Computing methodologies** → *Parallel computing methodologies*; • **Information systems** → **Computing platforms**; **Storage class memory**; **Graph-based database models**.

Keywords

evolving graphs, incremental computing, persistent/non-volatile memory

ACM Reference Format:
Yunmo Zhang, Jiacheng Huang, Xizhe Yin, Junqiao Qiu, Hong Xu, and Chun Jason Xue. 2025. PIE: Enabling Fast and Scalable Incremental Evolving Graph Analytics on Persistent Memory. In *2025 International Conference on Supercomputing (ICS '25), June 08–11, 2025, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3721145.3730419

1 Introduction

Real-world graphs that evolve continuously over time are ubiquitous in today's data-driven era [54]. These dynamic graphs are broadly employed across various domains, including social networks [6], machine learning [16], biological informatics [17], network monitoring [24] and among others.

*Corresponding Author

ICS '25, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-1537-2/25/06
https://doi.org/10.1145/3721145.3730419

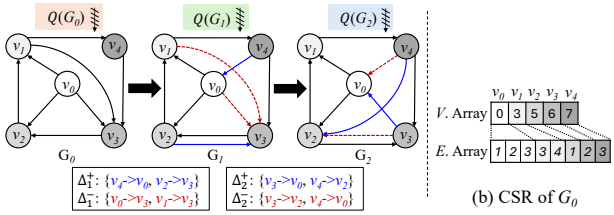
Effective analyses of such dynamic graphs are crucial for extracting timely insights and making informed decisions [54].

Evolving graph analytics is among the most popular analysis classes on dynamic graphs [15, 25, 45]. It provides real-time tracking of a graph property over a specific time window by computing values on *a sequence of* snapshots. For example, evolving graph analytics was used in an infectious disease monitoring platform for tracing epidemics spread over time during COVID-19 [67]. It was also used to extract the varying distance between two users in a social network (shortest-path query) [51], or measure the duration of a network failure (reachability query) [24].

As real-world graphs continue to grow in scale and the need to analyze more historical data intensifies, in-memory graph systems face a significant scalability challenge due to the limited DRAM capacity. Persistent memory (PM) technologies, such as non-volatile memory [18] and CXL-SSD [29], emerge with memory-like access semantics and an order of magnitude greater capacity at a cheaper per-gigabyte cost than DRAM. These features make PM well-suited for memory-intensive evolving graph processing [2, 8, 13, 14, 50]. Recent works, including DGAP [22] and XPGraph [63], highlight the potentials of PM in evolving graph processing.

Though existing PM-based evolving graph systems have achieved either fast ingestion of updates or efficient data access, none of them supports incremental analytics of evolving graphs. Incremental graph analytics has been proven to have superior efficiency in many in-memory graph systems [9, 43, 62], as evolving graphs typically exhibit slow, gradual changes over time, resulting in high similarity (often exceeding 99% [51]) between consecutive snapshots. For a series of snapshots, incremental analysis approaches leverage the results of a previous snapshot to compute the targeted snapshot, instead of re-computing the results from scratch for each snapshot. However, their efficiency diminishes when processing deleted edges [1, 12], incurring significant computations to align previous results with updates before re-converging on the new snapshot.

In this paper, we introduce **PIE**, a framework that enables fast and scalable Persistent memory based Incremental Evolving graph analytics. PIE is built upon the main idea of a recent evolving graph incremental analysis and representation work CommonGraph [1], which transforms the deletions into additions to avoid processing expensive deletions in traditional incremental analytics. Starting from evaluating the query on the common graph across snapshots, this approach incrementally reaches each snapshot with only additions to handle on a schedule that maximizes the sharing of analysis results. Our initial results show that a naive PM version of CommonGraph outperforms the traditional



(a) An evolving graph with 3 snapshots

(b) CSR of G_0

Figure 1: An illustrative example of an evolving graph and the CSR representation of its snapshot G_0.

incremental analysis approaches combined with state-of-the-art PM-based evolving graph representations. However, the transformation of deletions introduces extensive additional calculations and footprint storage on PM, leading to performance bottleneck and thus making a naive application of CommonGraph in PM far from optimal.

PIE incorporates a graph storage system and an analytics engine that work synergistically to prioritize the principle of maintaining low PM write and read amplification in supporting deletion-free incremental analysis. The storage system of PIE is based on the idea of separating the logical graph views required by incremental analysis and the physical storage of graph data. Exploiting our key observation on the relationship between graph data and graph views in analysis, we build a bitmap-based logical graph system, without requiring any PM writes. The storage system also includes a chunked neighbor index structure to reduce the PM reads needed to build the logical graph view of the common graph.

The analytics engine in PIE features a low building cost of deletion-free incremental analysis schedule based on the concept of the *Lasting Common Graph* (LCG), the maximal subgraph that remains across multiple snapshots till a timestamp. After refining the calculation methods of deletion-to-addition transformations, we identify a path between the common graph and snapshots that enables the reuse of multiple calculations around LCG. The LCG-driven incremental analytics engine of PIE thus conducts the query evaluation through a streamlined workflow that maintains the advantage of sharing analysis results in CommonGraph while reducing amplified read and calculation for transformation.

Finally, PIE introduces two optimizations to explore two key trade-offs in incremental analysis on PM, including a two-stage LCG-driven incremental analysis and caching the critical part of LCG to DRAM, to further improve performance under certain environments. Our evaluation of PIE using five commonly used graph queries on real-world graph datasets shows that PIE can significantly improve the performance efficiency and scalability of ingestion and analytics.

In summary, this paper made the following contributions:

- Presented the first work discussing PM-based evolving graph system that supports incremental analysis, including adapting different incremental analytics approaches to state-of-the-art in-PM graph formats, and revealing their key performance issues in ingesting updates and conducting analytics;
- Proposed a graph storage format that separates logical graph views with physical graph storage to achieve low amplified PM writes, as well as a chunked neighbor index technique to maintain low PM read amplification, for supporting incremental analysis;
- Designed a lightweight deletion-free incremental schedule that requires less calculation to build and two optimizations to exploit the trade-offs in the proposed framework to further improve performance;
- Compared PIE with the state-of-the-art systems through systematic evaluations. The results demonstrate that PIE brings significant performance improvements.

2 Background and Motivation

2.1 Persistent Memory

Persistent Memory (PM), including non-volatile memory [18] and CXL solutions such as CMM-H PM [55], offers durability and larger capacity at a more affordable cost than DRAM, while maintaining byte-addressability and comparable latency. These characteristics make PM an attractive option for a wide range of real-world applications [38], such as HPC [10, 11, 48, 64], database [31, 33, 72], and cloud service [30, 58]. Recently, PM has also gained attention for its potential in optimizing large-scale graph storage and processing [2, 8, 13, 14, 42, 50, 73].

Performance Characteristics. There are two key performance characteristics of PM, both in current technologies [27, 68] and future trends [29, 55, 59], crucial for building efficient systems upon PM. First, PM exhibits a noticeable performance gap with DRAM in general. DRAM's read and write bandwidths are 2~3× and 7~8× higher than PM's max read and write bandwidth [69], respectively. This gap is even more pronounced with CXL-SSDs [29, 57], which exhibit microsecond-level latencies. Second, small random accesses to PM experience significant bandwidth drop. Since data access is converted into a write/read at PM access granularity (hundreds of bytes to kilobytes), small accesses to PM may suffer from heavy read/write amplification [68, 70]. Modern PM systems use a limited buffer to merge adjacent small accesses, achieving sequential-like performance when their combined size matches the media granularity [68]. However, unmergeable small random accesses still suffer from substantial bandwidth loss. For instance, in PM with a 256-byte access granularity, 256-byte random writes offer 3~4× more bandwidth than 64-byte random writes, and even 40× than 4-byte writes, which are common in graph data storage.

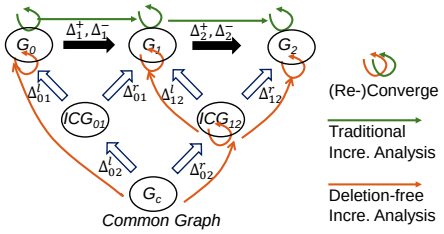


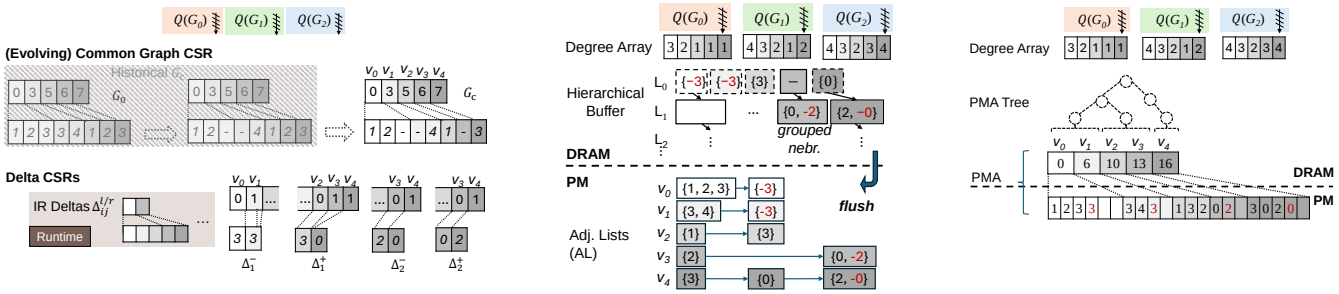
Figure 2: An incremental analysis schedule on the Triangle Grid in CommonGraph.

2.2 Evolving Graph Stores and Analytics

Existing dynamic graph systems can be classified into streaming graphs [7, 9, 43, 44, 62] and evolving graphs [1, 15, 25, 34, 37]. The former targets the latest snapshot of the graph as it continually changes, while the latter systems apply analytics over multiple snapshots within a time interval, providing insights or information drawn from historical data. This paper focuses on evolving graph systems. Formally, evolving graph analytics evaluates a query Q on a series of snapshots of an evolving graph G over a specified time period. At timestamp i, the graph is represented by a snapshot G_i. After applying a batch of edge updates [1], which consists of an addition batch Δ_{i+1}^+ and a deletion batch Δ_{i+1}^-, a new snapshot G_{i+1} is generated. An example is shown in Figure 1(a). The query (a graph algorithm), such as single-source-shortest-path (SSSP), is evaluated on each snapshot.

Evolving Graph Representations. A common approach to represent a snapshot is Compressed Sparse Row (CSR), which organizes vertex neighbors in an edge array with a vertex array for indexing, as illustrated in Figure 1(b). CSR provides space efficiency and locality but suffers from costly structural modifications during updates in dynamic graphs. To address this, amount of mutable CSR formats have been proposed. For instance, PCSR [65, 66] and VCSR [23] leverage the Packed Memory Array (**PMA**) to introduce space between different vertices' neighbors in the edge array. These spaces are managed by a binary tree, known as the PMA tree, which ensures neighbor density within a range. In addition, **Graphone** [34] extends traditional adjacency lists (AL) with multi-version degree arrays to manage updates across snapshots. The most recent in-DRAM evolving graph representation is **TgStore** [5], which stores the union of the base graph and updated edges as ordered edge lists, associating bitmaps for each edge to indicate if it exists in a snapshot.

[1]This paper focuses on edge updates. Weight updates can be managed by deleting the old edge and adding a new one. Vertex updates involve removing/adding all edges of the vertex.



(a) (In-DRAM/In-PM) CommonGraph (b) XPGraph (In-PM Graphone) (c) DGAP (In-PM PMA)

Figure 3: State-of-the-art Evolving Graph Frameworks: CommonGraph, XPGraph, and DGAP.

Meanwhile, graph analytics systems employs different formats to drive their incremental analysis. **Kickstarter** [62] and Graphbolt [43] utilize adjacency arrays (with offline graph updates ingestion), while Risgraph [9] enhances their formats with localized hash indices to accelerate update retrieval. **CommonGraph** [1] introduces a multi-CSR design, storing updated edges out of the base graph (G_0) to reduce structural modifications, as shown in Figure 3(a). It updates G_0 as a common graph by marking edges shown in deletion batches, and constructs intermediate deltas at runtime for deletion-free incremental analysis.

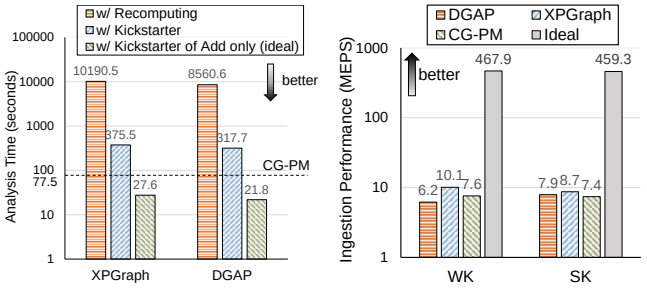
Evolving Graph Analytics. A straightforward approach is applying the query independently to each graph snapshot, known as *recomputation*. However, this often leads to redundant calculations, as evolving graphs change slowly over time relative to their sizes (usually less than 1%) [5, 25, 51, 60]. Therefore, a set of incremental analysis approaches, such as KickStarter and Graphbolt, have been proposed to *reuse* the analysis results from the previous snapshot when serving queries on new snapshots [9, 43, 60, 62]. They generally contain two phases: (P1) identifying and resetting analysis results affected by edge deletions in the previous snapshot; (P2) updating these results to account for edge additions, ultimately re-converging on the new snapshot.

Recently, **CommonGraph** [1] proposed a new incremental analytics design that transforms deletions into additions for processing. For a series of snapshots, it first analyzes their common (sub-)graph G_c, and then incrementally reaches each snapshot with only processing additions, i.e., P2 in traditional incremental analysis. For example, in Figure 2, snapshots G_0 and G_1 share a subgraph ICG_{01}. The results of G_0 (G_1) are derived by incrementally analyzing addition delta Δ^I_{01} (Δ^r_{01}) from the analysis results of ICG_{01}. In its work-sharing mode, CommonGraph builds the shared edge sets among multiple snapshots recursively, progressing through the intermediate common graphs (ICGs) to the final common graph G_c, creating a triangle grid (Figure 2). A *schedule* on triangle grid determines the route from G_c to each snapshot through ICGs, where the analysis results re-converged on ICGs are further shared across part of snapshots. Among various schedules, CommonGraph chooses the one with the smallest total delta sizes to maximize analysis results sharing. CommonGraph also has a basic mode called Direct Hop, which reaches each snapshot directly from G_c without sharing the results of ICGs, making it around 2× slower than work-sharing mode in PM according to our empirical observation. Thus, this paper focuses on the work-sharing mode.

PM-based Evolving Graph Processing. Recent PM-based graph systems highlight the need to handle evolving graphs, which is a memory-intensive case suitable for PM. The state-of-the-art (SOTA) PM-based evolving graph systems are XPGraph [63] and DGAP [22]. **XPGraph** is essentially a PM-based Graphone, utilizing AL to store the loaded graph and dynamic updates in PM, as shown in Figure 3(b). To reduce write amplification caused by the access granularity mismatch between 4-byte neighbor and 256-byte PM media, XPGraph buffers neighbors in a hierarchical DRAM pool before flushing coalesced writes to PM. When evaluating graph queries, XPGraph allows its analysis engine to access data from AL in both DRAM and PM. **DGAP** employs a PM version of PMA to improve read locality, as shown in Figure 3(c). DGAP places its degree array, PMA's vertex array and PMA tree in DRAM, while the edge array in PM, to boost ingestion and analytics efficiency.

2.3 Motivation

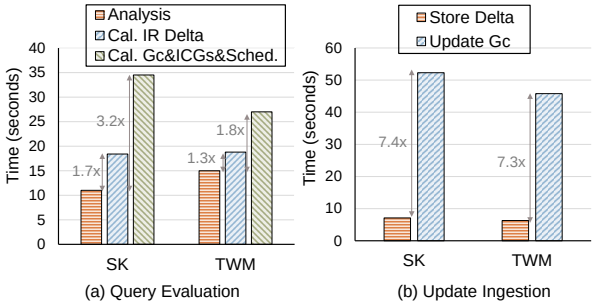
SOTA PM-based systems, XPGraph and DGAP, have shown promising results in update ingestion or snapshot access. However, both utilize recomputation for query evaluation, causing substantial redundant computations and missing out



(a) The cost of evaluating an SSSP query on SK [52].

(b) Ingestion performance of SOTA frameworks on WK [35] and SK [52].

Figure 4: Performance comparison among PM-based systems in query evaluation and update ingestion.



(a) Query Evaluation

(b) Update Ingestion

Figure 5: Breakdown of Analytics and Ingestion Performance of CG-PM on SK [52] and Twitter-MPI [21].

on the benefits offered by recent advancements in incremental processing techniques.

Potentials of Incremental Processing in PM. We first verify the benefits of moving existing incremental processing designs to PM. We integrate the incremental graph analytics scheme in Kickstarter into XPGraph and DGAP, and implement a PM-based adaptation of CommonGraph (referred to as CG-PM). The performance results of query evaluation and update ingestions over 12 snapshots are reported in Figure 4. More detailed dataset descriptions and experimental settings are provided in Section 5. We observe that incremental analytics approaches significantly outperform recomputation, achieving up to 27× faster when using the KickStarter scheme for mixed edge additions and deletions. Additionally, we find that deletion handling in PM-based incremental processing remains a critical bottleneck. In Figure 4(a), analyzing snapshots that involve only edge additions is 13-17× faster than processing those with both additions and deletions in the same amount. This validates that the primary motivation behind CommonGraph of transforming deletions to additions is also valid in PM. In fact, CG-PM already delivers up to 4.8× speedup than the KickStarter schemes. We thus focus on CommonGraph-like incremental evolving graph analytics in PM.

Limitations of PM-based CommonGraph. However, directly moving CommonGraph to PM does not yield the expected performance gains observed in DRAM due to fundamental differences in hardware characteristics and the lack of PM-aware optimizations in its design. Figure 4(a) shows that CG-PM is 3.6× slower than KickStarter that runs on the same amount of addition-only updates. This is because the overhead of transforming deletions into additions is particularly costly in PM. In addition to query evaluation, CG-PM also presents a significant gap with the ideal ingestion performance. As shown in Figure 4(b), despite achieving comparable ingestion performance to SOTA PM-based systems due to the contiguous data layout of multi-CSRs, CG-PM still falls short of an optimal ingestion rate by 50×. Note that the ideal case is about the sequential writes of flushing edge updates to an edge list representation in PM and it is not practical for subsequent graph analysis.

To illustrate the performance bottleneck of CG-PM, we break down its end-to-end time costs on two large graphs, as shown in Figure 5. When evaluating an SSSP query, the time required to construct the grid far exceeds that needed for analysis. Specifically, calculating intermediate (IR) deltas takes 1.3× to 1.7× longer, and calculating G_c, ICGs and the schedule takes 1.7× to 3.2× longer than executing the incremental analysis. These additional reads and calculations thus become the bottleneck of serving a query. Moreover, updating G_c with each new snapshot introduces around 7.3× more overhead than storing the ingested delta batch, significantly slowing down the dynamic graph ingestion.

3 The Design of PIE

3.1 Overview

Observing the great benefits introduced by CommonGraph-based incremental processing and the significant PM access inefficiency from its direct adoption, we develop PIE, a PM-based evolving graph storage and analytics system that enables fast and scalable deletion-free incremental analysis. Figure 6 provides the overview. At a high level, it consists of three components: a logical-view-decoupled graph storage system, a scheduling-free incremental graph analytics engine, and PM-friendly optimizations.

In the storage system (❶), PIE incorporates two techniques. (1) To avoid PM writes for building the triangle grid (TG) during ingestion and analytics runtime, we separate the logical graph view needed for analysis from the physical storage. This is achieved by using live bitmaps in DRAM to

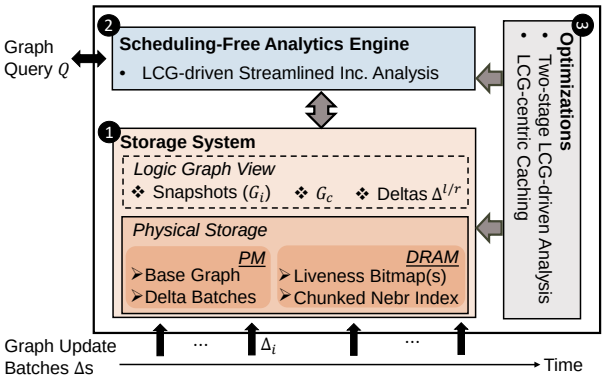


Figure 6: Overview of PIE.

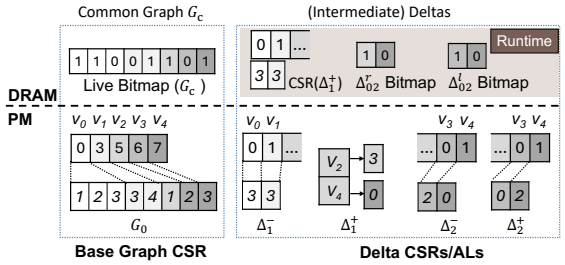


Figure 7: Evolving Graph Representation in PIE.

create the logical view of TG components (including snapshots) without modifying or adding to the graph data stored in PM. This design leverages our observation about the inclusion relationship between the TG components and the ingested graph data. (2) To reduce extensive and redundant PM reads in calculating the common graph, we introduce a chunked binary tree to index the neighbors of high-degree vertices in DRAM.

The analytics engine (❷) employs a scheduling-free approach that bypasses the complexities of handling deletions while achieving low transformation cost. Based on the observation that the schedule path through the Lasting Common Graphs (LCGs) can be computed with minimal cost, an LCG-driven streamlined incremental analysis engine is integrated.

Finally, PIE introduces two optimizations (❸) that explore the trade-offs in our design to further enhance performance. The first optimization is the two-stage LCG-driven incremental analysis, which explores the trade-offs between the construction overheads of intermediate shared graphs and the improved sharing of analysis results. The second optimization caches critical parts of the LCG in DRAM, exploiting the memory resource to further improve the performance of the analysis. Below, we present the first two components in detail, and the optimizations are introduced in Section 4.

3.2 Storage System

The graph representation is essential for efficient graph ingestion and retrieval. PIE absorbs the multi-CSR format of CommonGraph as its basic structure for delta batches, which compactly stores deltas separately from the base graph, offering several benefits for PM-based processing. First, the out-of-place contiguous storage of deltas allows for coalesced writes when stored to PM, eliminating small random writes for new edge data (a 4-byte destination vertex). Second, the compact CSR format of a delta enhances read efficiency during incremental analysis. Our empirical findings show that processing deltas—traversing their edges to update results—is a time-intensive step in deletion-free incremental analysis, often taking longer than re-convergence. The contiguous layout of delta edges improves the locality during traversal. However, this format becomes inefficient with small delta sizes, as the vertex array of delta CSR can far exceed the number of edges, leading to redundant indexes of empty neighbors. To resolve it, we enhance multi-CSR with a hybrid delta format. As shown in Figure 7, when the number of unique source vertices in a delta batch is below a threshold (256 in our work), we store the delta's adjacency list in PM and rebuild its CSR format in DRAM before query evaluation.

In PIE, all graph data, including the base graph CSR and delta CSRs, is entirely stored at PM, as shown in Figure 7. Based on this physically stored graph data, we build the triangle grid (TG) to support deletion-free incremental analysis *without* requiring additional PM writes, as we will discuss soon. Another key distinction between our format and CommonGraph is that the base graph CSR in our work remains unchanged from being stored to the next checking point to avoid small random PM writes.

The detached logical graph view of TG components. In addition to storing data, the graph system must provide access to the common graph G_c and intermediate (IR) deltas in the triangle grid to facilitate deletion-free incremental analysis. Intermediate common graphs (ICGs) and snapshots could be viewed as the combination of G_c and these deltas. However, as discussed in §2.3, updating G_c and representing IR deltas in PM are costly for both ingestion and analysis runtime. A potential solution is to store the IR deltas in DRAM, but this approach requires significant DRAM resources, limiting its feasibility for large-scale graphs or numerous snapshots, as N snapshots require N^2 IR deltas.

We propose to create *logical graph views* of TG components by utilizing our insights regarding the inclusion relationship between graph data and TG components. These logical views are detached from the physical graph storage, meaning that the graph itself is not actually stored when presenting its view. This is achieved through the use of bitmaps

in DRAM and by sharing the physical storage of graph data in PM across different logical views, as shown in Figure 7.

Denotation. We use $ICG_{i,j}$ to denote the intermediate common graph shared by consecutive snapshots $\{G_i, \cdots, G_j\}, i < j$. Note that $ICG_{0,N}$ is the common graph G_c and $ICG_{0,j}, j < N$, is a historical common graph. We use $\Delta^l_{i,j-1}$ to denote the intermediate delta between $ICG_{i,j}$ and its upper left graph $ICG_{i,j-1}$, referred to as *left intermediate delta*, while Δ^r_{ij} denotes the intermediate delta with its upper right graph $ICG_{i+1,j}$, referred to as *right intermediate delta*. An example of denotations is shown in Figure 2.

For the relations between IR deltas and the physically stored delta batches, we have the following claim.

Lemma 3.1 (**Delta Inclusions**). *Any edge in a left intermediate delta $\Delta^l_{i,j}$ is contained by the deletion delta Δ^-_j; while any edge in a right intermediate delta $\Delta^r_{i,j}$ is contained by the addition delta Δ^+_i.*

This inclusion relation could be naturally proved by our following computations of IR deltas:

$$\Delta^l_{ij} = \Delta^-_j - \sum_{k=i}^{j-1} \Delta^+_k; \quad \Delta^r_{i,j} = \Delta^+_i - \sum_{k=i+1}^{j} \Delta^-_j. \tag{1}$$

This implies a left (right) IR delta includes the edges from the deletion (addition) delta while removing its intersection with the previous (following) addition (deletion) deltas. Note that this computation of IR deltas differs in form from the recursive computation in CommonGraph by form, which uses $\Delta^r_{i,j-1}$ and $\Delta^l_{i+1,j}$ to generate $\Delta^r_{i,j}$, as summarized in Figure 10. Yet both computations yield equivalent results (which can be proved easily). With this inclusion relation, we build a bitmap $bp^l_{i,j}(bp^r_{i,j})$ in DRAM associated with the edge array of $\Delta^-_j(\Delta^+_i)$ to logically represent $\Delta^l_{i,j}(\Delta^r_{i,j})$. In this bitmap, a bit is False when the edge in the delta batch is not contained by the IR delta.

The (historical) common graphs have a similar inclusion relation with the physically stored base graph.

Lemma 3.2 (**Common Graph Inclusion**). *Any edge in a (historical) common graph $ICG_{0,j}$ is contained by the base graph G_0.*

This claim is obvious given the following computation:

$$ICG_{0,j} = G_0 - \sum_{k=1}^{j} \Delta^-_k. \tag{2}$$

In other words, a (historical) common graph incorporates the edges of G_0 while excluding any edges that intersect with the deletion delta batch prior to its timestamp. Again, this computation yields an equivalent result as the recursive computation in CommonGraph (Figure 10). With this relation, G_c could be logically represented using a bitmap bp^c in DRAM associated with the edge array of G_0, where a bit is False when an edge from the base graph does not exist in the common graph. With this bitmap, PIE resets the bit at the position corresponding to the deleted edge in G_0 upon each ingested deletion delta, thereby avoiding small random writes associated with changing G_0 CSR in PM.

Table 1: The Graph Abstractions provided by PIE.

Logical Graph	Physical Structure
$\Delta^l_{i,j}$	Filtering Δ^-_j through $bp^l_{i,j}$
$\Delta^r_{i,j}$	Filtering Δ^+_i through $bp^r_{i,j}$
G_c	Filtering G_0 through bp^c
$ICG_{i,j}/G_i$	Combining G_c and intermediate deltas

Table 1 summarizes the mapping from the physical multi-CSR graph storage to the logical graph view via bitmaps in PIE. Exploiting this detached logical graph view, PIE avoids extra writes to PM, aside from the graph data, during ingestion and deletion-free incremental analysis runtime.

The chunked neighbor index. Since calculating the evolution of the common graph (G_c) contributes to the majority of amplified reads in CG-PM, as shown in Section 2.3, we design an index structure for G_c to accelerate its calculation. Specifically, the G_c update is computed by N times of set-set Difference/Intersection operations between G_0 and each of N deletion deltas, necessitating either binary or linear searches over G_0. This process induces a considerable number of redundant reads from the base graph.

An intuitive approach to mitigate this read amplification is to construct an index for the neighbors in the base graph. Risgraph [9] creates a hash table for the neighbors in its adjacency list, enabling $O(1)$ edge lookup. However, directly indexing the neighbors requires substantial DRAM usage, which is at odds with the scenario this paper addresses, where the graph edge array cannot fit into the memory.

Exploiting the trait of PM access granularity, we build a *chunked* neighbor index for the high-degree vertices in the base graph to accelerate the calculation of the common graph while maintaining a low DRAM resource usage. As shown in Figure 8, the neighbors of a high-degree vertex are segmented into 256-byte chunks, aligning with the media granularity of existing PM [18]. Then, a binary tree is built upon the pivot value of each chunk as the leaf node, e.g., the largest value in the chunk. The chunk size could be set by users as the buffer size of their PM device. The height of the tree is then $O(log(deg(v)/size(buf)))$. This binary tree is stored as an array on DRAM, in which the position of a leaf node in the array directly indicates the position of the chunk containing the target in the edge array. With this index, each edge from the deletion delta looking up the base graph edges requires at most one PM media load.

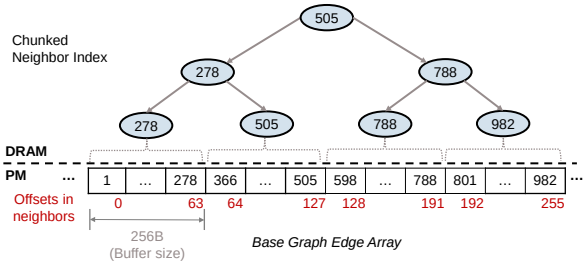


Figure 8: The chunked neighbor index for a high-degree vertex in the base graph.

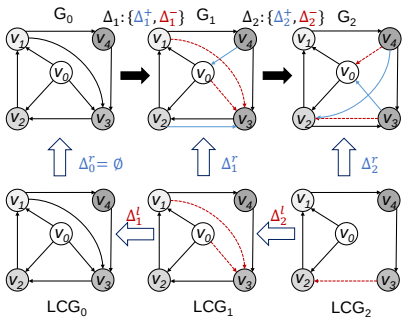


Figure 9: Streamlined incremental analysis.

	CommonGraph	PIE
Common Graph $ICG_{0,j}/LCG_j$	$ICG_{0,j-1} - \Delta_j^-$	$LCG_{j-1} - LCG_{j-1} \cap \Delta_j^-$
Left Delta $\Delta_{i,j}^l/\Delta_j^l$	$\Delta_j^-, \quad j-i=1$ $\Delta_{i+1,j}^l - \Delta_{i,j-1}^r, j-i>1$	$LCG_{j-1} \cap \Delta_j^-$
Right Delta $\Delta_{i,j}^r$	$\Delta_i^+, \quad j-i=1$ $\Delta_{i,j-1}^r - \Delta_{i+1,j}^l, j-i>1$	$\Delta_i^+, \quad j-i=1$ $\Delta_{i,j-1}^r - \Delta_j^-, j-i>1$

Figure 10: Calculations of components required in the deletion-free incremental analysis of two systems.

The chunked neighbor index is constructed before the ingestion to facilitate the calculation of evolving G_c and remains unchanged until released after dynamic ingestion complete. Its memory footprint in DRAM depends on the high-degree vertex threshold set by users, but it is generally minimal. In this work, we set the threshold at 256 neighbors, which identifies 1% to 1.3% vertices as high-degree, consuming only 0.9% to 1.2% of PIE's entire DRAM usage across the graphs evaluated in this work (§5). Furthermore, the index construction can be seamlessly integrated into the parallel creation or re-creation of the base graph CSR. During this process, threads store neighbors to the edge array according to the vertex index (prefix sum) and select a pivot value upon reaching a chunk boundary.

Crash Consistency and Recovery. PIE ensures crash consistency by atomically updating a snapshot number metadata after persisting a new update batch. Specifically, an `sfence` is issued between the delta batch flush and the snapshot number update. This guarantees that, after a crash, queries can access consistent views of snapshots committed before the crash, avoiding any partial data. Then rolling back to a consistent state involves deleting delta CSRs with IDs larger than the persisted snapshot number. The crash consistency level provided by PIE is thus *buffered durable linearizability* [26], which is sufficient to enable recoverability of evolving graph systems, i.e., PIE can rebuild TG components post-crash. For instance, G_c can be rebuilt by reconstructing its bitmap by scanning CSRs of the base graph and all valid deletion deltas.

3.3 LCG-driven Incremental Computations

To maximize the analysis results sharing, CommonGraph first constructs a complete triangle grid and then computes a schedule that approximately minimizes the delta size along the path [1]. However, this method incurs significant overhead from grid construction, as discussed in Section 2.3. In contrast, the analytics engine in PIE prioritizes minimal scheduling and construction costs for deletion-free incremental analysis while maintaining the analysis results sharing.

We introduce the concept of the *Lasting Common Graph* (LCG), which are defined as the historical common graphs in TG, i.e., $LCG_j = ICG_{0,j}$. Each LCG_j corresponds to a snapshot G_j and represents the common graph of the snapshots till the time j. PIE builds its deletion-free incremental analysis schedule centering around LCG, named as *LCG-driven incremental analysis*. As shown in Figure 9, this analytics approach on an LCG grid first analyzes the latest LCG, i.e., G_c, same as CommonGraph, and then reaches the preceding LCGs recursively before incrementally reaching each snapshot. In this process, the analysis results first converge on LCGs and then are reused by the incremental analysis of the corresponding snapshot and its preceding LCG. Specifically, the delta between LCGs, denoted by Δ_j^l, is the leftmost IR delta in TG generated at CommonGraph; while the delta between LCG_j and its corresponding snapshot G_j, referred to as *completion delta* and denoted by Δ_j^r, comprises a set of right IR deltas in TG, as shown in Equation 3.

$$\Delta_j^l = \Delta_{0,j}^l; \qquad \Delta_j^r = \sum_{k=0}^{j-1} \Delta_{k,j}^r. \tag{3}$$

The advantage of the LCG-driven approach is that the construction cost of the LCG grid is as small as the cost of generating consistent snapshots, which are essential for dynamic graph analytics. Building N consistent snapshots in a dynamic graph system entails applying N deletion batches to previously inserted edges, including the base graph and any prior addition batches. In this process, each deletion delta Δ_i^- requires a Difference/Intersection set-set operation

with G_0 and same operations with Δ_k^+ for $k < i$. LCG-driven analysis demands the same complexity of set-set operations to create the LCG grid. As shown in Figure 10, in PIE, each deletion delta Δ_j^- needs a Difference/Intersection operation with LCG_{j-1} to compute the common graph (note that LCG_j is the subset of G_0), and same operations with $\Delta_{i,j-1}^r$ for $i < j$ to compute the right deltas needed in the grid (note that $\Delta_{i,j-1}^r$ is the subset of Δ_i^+). The left deltas can reuse the calculations from LCGs, requiring no additional calculation.

Our key observation to enable this minimal construction cost schedule is the reuse of calculations on LCGs when calculating a set of special left IR deltas Δ_j^l. Given the calculation method of $ICG_{0,j}$ in CommonGraph, our calculation method of LCG_j shown in Figure 10 is obvious; while $\Delta_j^l = LCG_{j-1} \cap \Delta_j^-$ because $LCG_{j-1} \cap \Delta_j^-$ is the complement of LCG_{j-1} to LCG_j, which is the definition of Δ_j^l. With this calculation method, during the calculation of the common graph LCG_j, each edge found both in LCG_{j-1} and Δ_i^- should not only be removed from LCG_{j-1} but also added to Δ_j^l. This is implemented by setting the edge bit in $bp_{0,j}^l$ in addition to resetting its bit in bp_c. In this way, the calculation of Δ_j^l is merged into the procedure of generating the common graph. Additionally, we further refine the calculations of the right IR delta $\Delta_{i,j}^r = \Delta_{i,j-1}^r - \Delta_j^-$, by reusing the calculation Δ_j^l again, to $\Delta_{i,j-1}^r - (\Delta_j^- - \Delta_j^l)$. This reduces the complexity of a set-set operation by only computing the $\Delta_j^- - \Delta_j^l$ with the right delta. Since Δ_j^l is presented by $bp_{0,j}^l$ associated with Δ_j^-, $\Delta_j^- - \Delta_j^l$ requires no computations, as it can be accessed from the edges in Δ_j^- that do not pass the filtering of $bp_{0,j}^l$.

In summary, LCG-driven incremental analysis adopts a scheduling-free strategy that achieves the minimal cost of constructing the intermediate components needed in deletion-free incremental analysis.

4 Other Optimizations

4.1 Two-Stage Incremental Analysis

While LCG-driven analytics engine bases its incremental schedule on the optimal path with respect to the low-cost construction, it comes with the disadvantage of the increasing divergence between LCGs and their corresponding snapshot when the number of snapshots grows. This divergence diminishes the efficiency of incremental analysis. To mitigate this issue, we enhance the analytics engine with an optional technique that may introduce a few delta calculation costs but improve the sharing of analysis results.

We observe that the completion deltas largely overlap, particularly consecutive ones. As presented in Figure 10, the right delta $\Delta_{i,j}^r$, a component of a completion delta Δ_j^r, shared many edges with $\Delta_{i,j-1}^r$, a component of Δ_{j-1}^r. However, the LCG-driven analysis fails to effectively exploit this overlap, as each completion delta is processed independently to avoid the overhead of calculating left IR deltas that cannot be merged into LCG's calculation.

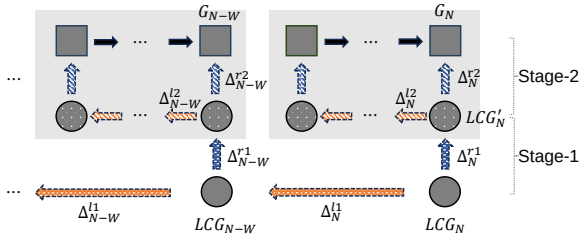


Figure 11: Two-stage incremental analysis. The second stage involves a sub-time window with size W.

To solve this potential issue, we extend the LCG-driven approach to two stages, with the first stage driven by a *local LCG* within a sub-time window, while the second stage zooming out to analyze the LCGs between sub-time windows. As shown in Figure 11, a local LCG, referred to as LCG'_N, is incrementally reached from LCG_N and its results will be shared by the snapshots in the sub-time window $\{G_{N-W+1}, \ldots, G_N\}$ during the second stage analysis. This local LCG facilitates the sharing of incremental analysis of consecutive completion deltas, reducing the divergence between LCG and the snapshot. In this two-stage grid, all deltas, except the local left IR Δ_i^{l2}, could directly use the calculation and storage introduced in §3.3. For instance, the sub-completion delta in the first stage Δ_i^{r1}, could be computed by $\sum_{k=1}^{i-w} \Delta_{k,i}^r$. Yet the local left IR deltas are new calculations (if $W > 2$): using Eq. 1, $\Delta_i^{l2} = \Delta_i^- - \sum_{j=i-W+1}^{i-1} \Delta_j^+$ $(i > W)$. In two-stage approach, N snapshots require $N - W$ extra delta calculations, which is still conservative compared with CommonGraph.

Selection of Sub-Time Window Size. The sub-time window size W is crucial for the two-stage approach to balance redundancy reduced and calculation introduced. A large W allows more snapshots to share the overlap of completion deltas but increases the overhead of calculating local left intermediate deltas. Conversely, a small W presents the opposite challenge. The optimal W depends on multiple factors, such as the size and structure of the evolving graph, the number and size of the deltas, and the specific graph query. In this paper, we empirically select $W = 4$.

4.2 LCG-centric Caching

As introduced in §3.2, PIE's storage systems store all graph data (multiple CSRs) in PM by default, allowing the system to scale for processing larger graphs while conserving memory resources [37, 53]. However, when memory spaces are

available, DRAM can be utilized to further accelerate query evaluation due to the noticeable performance gap between DRAM and PM. Unlike XPGraph, which caches the latest deltas in DRAM, PIE caches the critical part of the base graph G_0 for two reasons. First, the base graph, logically LCG, is needed at least twice during the LCG-driven incremental analysis of each snapshot (§3.3)–once for re-convergence on itself and once for re-convergence on the snapshot. Second, most accesses to the LCG are random, whereas deltas are accessed sequentially during incremental analysis in the multi-CSR graph format. To ensure efficient caching, we assign different cache priorities to various portions of G_0 CSR. The highest priority is given to the vertex array, followed by the neighbors of high-degree vertices in the edge array (set through a threshold by users), and finally to other vertices in the edge array. The selection of the critical parts ultimately depends on the memory budget. By default, PIE caches the vertex array of G_0 to maintain memory space efficiency.

5 Evaluation

We implement the prototyped PIE framework in C++ with using OpenMP for efficient parallelization. The framework consists of a library providing graph view interfaces introduced in Table 1. The analytics engine and storage system are developed on top of the GAP framework [3] and KickStarter [43, 62]. The PM-related access and memory management is developed using PMDK 1.12 [20].

5.1 Evaluation Methodology

Evaluated Systems. We compare PIE with the following evolving graph frameworks in the evaluation:

- Two state-of-the-art PM-based evolving graph frameworks, **XPGraph** [63] and **DGAP** [22]. For update ingestion, both frameworks handle deletion by inserting a new edge with the first bit of the destination vertex ID flipped. To ensure consistent versioning, XPGraph and DGAP read the neighbors into a DRAM array and apply the deletion to them before returning the consistent neighbors. We configure XPGraph and DGAP with their settings in their paper. For query evaluation, both frameworks originally used recomputation as their analytics engines. We incorporate an incremental analysis approach Kickstarter [43, 62] to these SOTA frameworks, denoted as **DGAP+KS** and **XPGraph+KS**, as their PM-aware analytics engine. All implementations are based on their publicly available source codes[2,3,4].

[2] https://github.com/ISCS-ZJU/XPGraph
[3] https://github.com/DIR-LAB/DGAP
[4] https://github.com/pdclab/graphbolt

Table 2: Graph Statistics.

Graph	V	E	Avg. Deg.	Dia.
WikiLinks (WK) [35]	13M	669M	64.3	12
uk-2005 (UK) [52]	39M	1.6B	47.4	23
It-2004 (IT) [52]	41M	2.1B	55.7	19
Twitter-2010 (TW) [36]	61M	2.4B	40.5	18
SK [52]	50M	3.7B	77.0	18
Twitter-MPI (TWM) [21]	53M	3.2B	74.7	18

- A direct adaptation of CommonGraph [1] in PM (**CG-PM**) with our best efforts. Their entire graph representations, such as multi-CSR representations of edge update batches, are retained in PM.
- A direct PM adoption of **TgStore** [5]. As TgStore primarily focuses on storage cost optimization rather than query execution, we only evaluate its update ingestion performance. Our implementations are based on our best understanding of the original paper.

Evaluation Platform. All experiments are conducted on a server with an Intel Xeon Gold 5317 3.00GHz processor with 12 physical cores. It has 8 DRAM DIMMs with 16 GB each (128GB in total) and 8 Optane DC DIMMs with 126GB each (1TB in total) in App Direct mode. The machine runs on Ubuntu 20.04.6 with Linux kernel 5.1.0. All source codes are compiled by gcc 10.5.0 with "-O3" flag.

Graph Datasets. Evaluations were conducted with a set of real-world graphs from different domains. Table 2 lists these graphs and their key characteristics. Following prior studies [22, 41, 60, 62], we initialize and warm up the system by preloading 50% of each dataset as the base snapshot (G_0) before benchmarking update ingestion. Unless otherwise specified, all experimental results presented in this section are obtained over 12 snapshots, where each snapshot involves 0.5% edge additions (randomly selected from the remaining dataset) and 0.5% edge deletions (randomly selected from the loaded portion). Both ingestion and query evaluation experiments are conducted with using 12 threads by default.

Graph Algorithms. We use five graph algorithms listed in Table 3 to evaluate the graph analysis performance as related work [1, 62].

5.2 Graph Ingestion Performance

Figure 12 presents the geometric mean ingestion throughput of PIE and four other systems in Million Edges Per Second (MEPS) across multiple datasets. PIE consistently outperforms all other evaluated systems, achieving speedups of 8.4×, 10.8×, 10.9×, and 52.5× over XPGraph, DGAP, CG-PM, and TgStore, respectively. These results demonstrate the effectiveness of PIE 's detached logical graph view and its contiguous layout design in maximizing data coalescing and reducing redundant writes. Although CG-PM also utilizes

Table 3: Graph Algorithms and their Edge Functions.

Algorithm	Push operation for an edge (s, d)
BFS	level(d) = min{ level(d), level(s) + 1; } **if** level(d) changed **then** add d to frontier;
SSSP	dist(d) = min{ dist(d), dist(s) + w(s, d); } **if** dist(d) changed **then** add d to frontier;
SSWP	wide(d) = max{ wide(d), min{ wide(s), w(s, d); } } **if** wide(d) changed **then** add d to frontier;
SSNP	narrow(d) = min{ narrow(d), max{ narrow(s), w(s, d); } } **if** narrow(d) changed **then** add d to frontier;
Viterbi	vite(d) = max{ vite(d), vite(s) / w(s, d); } **if** vite(d) changed **then** add d to frontier;

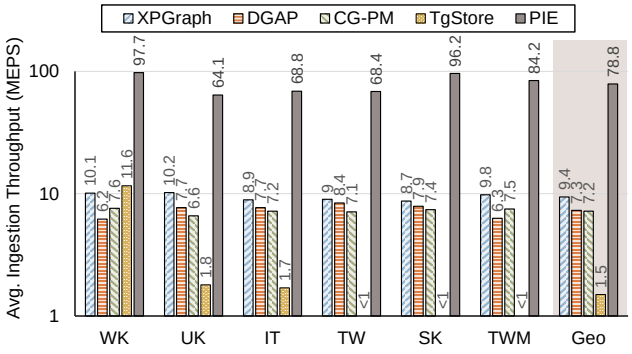


Figure 12: The Overall Performance of Ingestion (Larger the better).

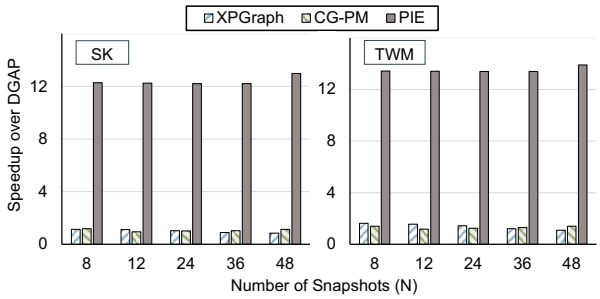


Figure 13: Effect of number of snapshots.

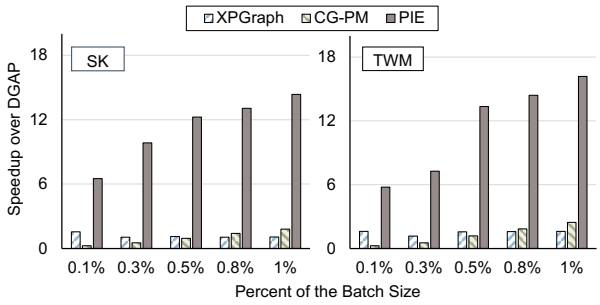


Figure 14: Effect of batch size.

a contiguous layout, the G_c updates introduce a large number of small random writes, and thus offsets its intended benefits. Compared to XPGraph and DGAP, which employ PM-aware storage optimizations, PIE achieves superior performance by reducing write amplification more effectively. XPGraph benefits from intra-vertex neighbor coalescing, providing an advantage over DGAP. However, its performance is constrained by the overhead of sorting and hierarchical buffering. TgStore shows a less efficient performance than other frameworks on large graphs, as it introduces substantial overhead from scans and writes across data blocks of both baseline snapshots and incremental edge data.

Scalability. We further evaluate the storage system's sensitivity to workload changes regarding the number of snapshots and batch size. We first vary the number of snapshots (N) from 8 to 48 while keeping a delta size at 0.5%. The results in Figure 13 indicate that PIE consistently outperforms the related work as N grows, confirming its stable ingestion performance benefits regardless of N. We then vary the delta sizes from 0.1% to 1% while keeping N at 12. The results in Figure 14 show that the speedups of PIE and CG-PM increase with larger delta sizes, as DGAP experiences longer blocking for rebalance under larger batch sizes.

5.3 Graph Analysis performance

Table 4 reports the overall query evaluation performance of evaluated frameworks, with PIE configured with all optimizations introduced in Section 4. The results show that PIE outperforms the compared frameworks by factors ranging from 3.9× to 105.9×. On average, PIE achieves 20.4×, 18.1×, and 6.4× speedups compared with XPGraph+KS, DGAP+KS, and CG-PM, respectively. We observe that the performance gap between PIE and the other systems is less evident on the smallest dataset (i.e., WK), but becomes increasingly significant on large power-law graphs such as UK and TWM. This is primarily due to the increasing overhead associated with identifying and propagating deletion effects over large highly skewed graph when applying traditional incremental approaches like KickStarter. In contrast, the deletion-free streamlined incremental analysis used in PIE is less sensitive to structural variations. Across diverse datasets and algorithms, PIE achieves 4.8× to 10.2× speedups over CG-PM, showing the effectiveness of its LCG-driven streamlined incremental analysis in mitigating redundant computations and optimizing PM access patterns.

Scalability. We also examine how well the analytics engine can handle varying numbers of snapshots and batch sizes. We first vary the number of batches from 8 to 48 while keeping the batch size at 0.5%. The results in Figure 15 show that

Table 4: Time of Executing Queries on Evolving Graphs.

Alg.	Graph	XPGraph+KS	DGAP+KS	CG-PM	PIE
BFS	WK	14.4s (6.4×)	14.1s (6.3×)	11.8s (5.2×)	2.2s
	UK	644.9s (88.0×)	222.6 (30.4×)	66.3s (9.0×)	7.3s
	IT	191.8s (29.0×)	78.4s (11.9×)	43.8s (6.6×)	6.6s
	TW	171.3s (17.8×)	247.1s (25.7×)	52.8s (5.5×)	9.6s
	SK	362.1s (37.5×)	152.9s (15.8×)	77.9s (8.1×)	9.7s
	TWM	549.1s (46.2×)	868.3s (73.0×)	74.5s (6.3×)	11.9s
SSSP	WK	19.8s (8.3×)	15.1s (6.3×)	11.5s (4.8×)	2.4s
	UK	553.1s (77.6×)	560.1s (78.6×)	64.6s (9.1×)	7.1s
	IT	191.4s (25.1×)	130.2s (17.0×)	43.7s (5.7×)	7.6s
	TW	163.8s (15.3×)	222.1s (20.7×)	54.0s (5.0×)	10.7s
	SK	375.5s (31.9×)	317.7s (27.0×)	77.5s (6.6×)	11.8s
	TWM	844.5s (55.2×)	797.2s (52.1×)	77.1 (5.0×)	15.3s
SSWP	WK	12.8s (5.2×)	11.6s (4.7×)	12.1s (4.9×)	2.4s
	UK	163.5s (21.5×)	248.6s (32.6×)	67.4s (8.8×)	7.6s
	IT	106.6s (15.7×)	85.4s (12.5×)	43.2s (6.3×)	6.8s
	TW	139.5s (13.4×)	209.0s (20.1×)	55.9s (5.4×)	10.4s
	SK	87.7s (8.4×)	101.5s (9.7×)	78.9s (7.5×)	10.5s
	TWM	287.5s (24.1×)	226.4s (19.0×)	76.0s (6.4×)	11.9s
SSNP	WK	12.1s (5.0×)	12.2s (5.0×)	12.3s (5.1×)	2.4s
	UK	648s (105.9×)	311.8s (51.0×)	62s (10.1×)	6.1s
	IT	199.1s (29.8×)	85.0s (12.7×)	43.4s (6.5×)	6.7s
	TW	69.6s (7.0×)	257.0s (25.8×)	55.1s (5.5×)	10.0s
	SK	162.8s (15.9×)	145.6s (14.2×)	78.5s (7.6×)	10.3s
	TWM	1031s (90.3×)	344.1s (30.1×)	75.2s (6.6×)	11.4
Viterbi	WK	9.7s (3.9×)	12.9s (5.2×)	12.3s (5.0×)	2.5s
	UK	93.3s (15.1×)	105.1s (17.0×)	63s (10.2×)	6.2s
	IT	163.1s (24.9×)	90.3s (13.8×)	43.4s (6.6×)	6.6s
	TW	152.8s (14.5×)	194.2s (18.4×)	56.3s (5.3×)	10.6s
	SK	121.9s (12.2×)	226.2s (22.6×)	78.8s (7.9×)	10.0s
	TWM	586.5s (42.7×)	374.4s (27.2×)	77.0s (5.6×)	13.7s
Norm. GMean		**20.4×**	**18.1×**	**6.4×**	1

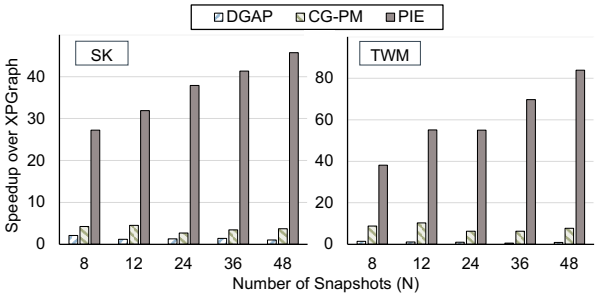


Figure 15: Effect of number of snapshots (SSSP).

PIE consistently outperforms other works, with the speedup increasing as N grows. In contrast, CG-PM experiences decreased speedup with more snapshots due to its quadratic complexity to N in grid construction. We then vary the batch size from 0.1% to 1% while keeping 12 snapshots. Figure 16 shows PIE and CG-PM achieve a consistent trend of increasing speedups due to the growing benefits of deletion-free incremental analysis.

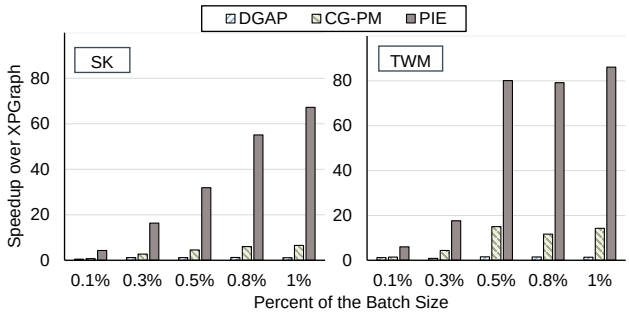


Figure 16: Effect of batch size (SSSP).

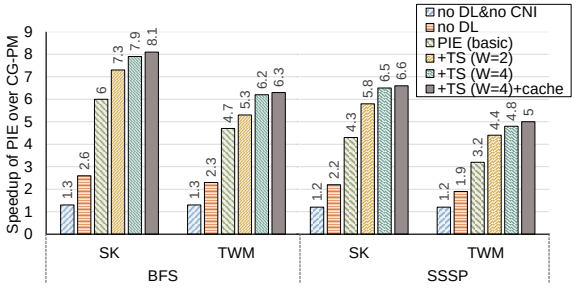


Figure 17: Impact of PIE Components.

5.4 Breakdown Analysis

To assess the contribution of the proposed PIE components to the final results in Table 4, we conduct a breakdown analysis by removing the Detached Logical graph view (no DL) and the Chunked Neighbor Index (no CNI) from PIE-Basic, presented in §3. We then add the two-stage technique (TS) with varying sub-time window sizes and DRAM caching of the G_0 vertex array (cache). The results in Figure 17 show that PIE-Basic, including DL, CNI and LCG-driven approach, achieves average speedups of 5.3× for BFS and 13.7× for SSSP over CG-PM. Among the three components, DL contributes the most benefits by eliminating runtime PM writes. The two-stage technique reduces the analysis time of PIE-Basic by an average of 14.0% for BFS and 21.8% for SSSP when $W = 2$, and by 24.1% for BFS and 29.3% for SSSP when $W = 4$. Finally, adding DRAM caching further reduces analysis time by 1.5% to 3%.

5.5 Recovery Cost and DRAM Usage

Table 5 reports the time costs of recovering the system status of three systems after a crash, which happens after ingesting 12 batches and before analytics. PIE and DGAP could recover the status within 10 seconds, as their recoveries only involve a scan of the edge array to recompute the metadata. XPGraph needs to scan the log to rebuild the graph data in AL, which is significantly slower. We also profile the peak DRAM space

Table 5: Recovery Cost (Seconds).

	WK	UK	IT	TW	SK	TWM
XPGraph	3.3	13.8	17.6	16.8	24.6	45.7
DGAP	1.6	4.2	5.0	5.8	8.3	7.7
PIE	0.8	2.2	2.4	3.1	4.5	3.9

Table 6: Peak DRAM Usage.

	XPGraph	DGAP	PIE w/o Cache	PIE w/ Cache
WK	17.4GB	2.0GB	686.2MB	741.2MB
TW	21.8GB	9.4GB	2.8GB	3.2GB
SK	20.6GB	7.1GB	2.2GB	2.5GB

occupation of three systems during the entire process of ingesting and analyzing 12 batches. The results reported in Table 6 show that XPGraph requires the most DRAM space as it needs to store a 16GB buffer pool and extensive metadata in DRAM. In contrast, PIE requires minimal DRAM resources, even after applying the DRAM caching optimization.

6 Discussion of Generality

The PIE framework is currently implemented on Intel Optane DCPMMs. Despite Intel's discontinuation of Optane PM in July 2022 [19], research in PM technologies continues due to their great potential for data-intensive applications. Meanwhile, upcoming PM products for the new generation Intel Xeon Scalable processors (e.g., Sapphire Rapids) will retain 3D XPoint as their underlying media [39], ensuring continued applicability of PIE. Beyond Intel, next-generation PM technologies using alternative media are emerging to redefine future memory and storage architectures [4, 49, 59]. They are expected to share similar architectural constraints, including limited internal buffering and high media access granularity. The design of PIE can be generalized to other PM devices that exhibit access granularity mismatches, such as CXL-based Memory-Semantic SSDs [56] and KIOXIA's XL-FLASH [32], which have block-level access granularity (e.g., 4 KB flash pages) significantly larger than cache-line sizes, leading to severe write amplification. As a result, PIE will remain highly relevant and is expected to be effective for PM-based evolving graph analytics across a variety of next-generation PM architectures.

7 Related Work

PM-based Graph Systems. Several proposals have investigated the design of static graph systems using single-machine non-volatile memory. Sage [8] designs a semi-asymmetric model that combines PM and DRAM to minimize costly PM writes. Bae et al. [2] modifies Ligra [61] to utilize PM in *memory* mode, controlling PM through the storage stack. Ghosh et al. [13] and Peng et al. [50] examine graph applications on PM in *App Direct* mode, highlighting its competitive performance against DRAM and memory mode. PIE also targets App Direct mode, enabling direct access to durable data via load/store. For dynamic graphs on PM, an early work is NVGraph [40], which employs an AL-based graph structure, with each vertex having an array of pointers to index neighbors in different snapshots. Since NVGraph was developed before any PM device was released, recent work [22, 63] does not compare with NVGraph for a fair comparison [22].

Evolving Graph Systems. On the graph representation front, Aspen [7] and TEGRA [25] exploit function trees to represent evolving graphs, indexing the immutable snapshots via pointers and storing the mutated sections out-of-place with pointers to these sections updated in the tree. Version traveler [28], LLAMA [41] and Graphone [34] employ linked-based format to manage vertex neighbors with the degree-array-like structures for identifying snapshots. Terrace [47] proposes a hierarchical structure that stores low-degree vertices in an array for better locality and high-degree vertices in tree structures for efficient updates separately. On the analytics front, several in-memory incremental analysis systems have emerged [43, 60, 71]. Early approaches such as Tornado [60] and Naiad [46] support only edge additions, while recent work, Kickstarter [62] and Graphbolt [43], includes deletions, with Graphbolt further supporting the Bulk Synchronous Parallel (BSP) semantics. IncBoost [71] discusses the efficiency of different runtime data representations in Kickstarter, which we adopt the same in PIE as related systems to ensure a fair comparison.

8 Conclusion

This paper proposes PIE, a PM-based evolving graph system that features efficient deletion-free incremental analysis of multiple snapshots. For the amplified write, read, and calculations introduced by the existing deletion-free incremental analysis approach, PIE presents the detached logical graph view, chunked neighbor index, and LCG-driven incremental analysis to resolve them, respectively. Our comprehensive evaluation confirms that PIE outperforms state-of-the-art incremental analysis and PM-based graph systems in ingestion, analysis, recovery, and DRAM usage.

Acknowledgments

We are grateful to the anonymous reviewers for their constructive comments and suggestions. The work was supported in part by City University of Hong Kong internal and donation fundings (No. 9610598 and No. 9220148), and the National Science Foundation (NSF) Grant (No. 2105006).

References

[1] Mahbod Afarin, Chao Gao, Shafiur Rahman, Nael Abu-Ghazaleh, and Rajiv Gupta. 2023. Commongraph: Graph analytics on evolving data. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[2] Hanyeoreum Bae, Miryeong Kwon, Donghyun Gouk, Sanghyun Han, Sungjoon Koh, Changrim Lee, Dongchul Park, and Myoungsoo Jung. 2021. Empirical guide to use of persistent memory for large-scale in-memory graph analysis. In *IEEE International Conference on Computer Design (ICCD)*.

[3] Scott Beamer, Krste Asanović, and David Patterson. 2017. The GAP Benchmark Suite. arXiv:1508.03619 [cs.DC]

[4] Nantero Bill Gervasi. 2022. A Persistent CXL Memory Module with DRAM Performance, In Storage Developer Conference (SDC), SNIA. https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/overview.html.

[5] Yongli Cheng, Yan Ma, Hong Jiang, Lingfang Zeng, Fang Wang, Xianghao Xu, and Yuhang Wu. 2024. TgStore: An Efficient Storage System for Large Time-Evolving Graphs. *IEEE Transactions on Big Data* (2024).

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

[7] Laxman Dhulipala, Guy E Blelloch, and Julian Shun. 2019. Low-latency graph streaming using compressed purely-functional trees. In *Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*.

[8] Laxman Dhulipala, Charles McGuffey, Hongbo Kang, Yan Gu3 Guy E Blelloch, Phillip B Gibbons, and Julian Shun. 2020. Sage: Parallel Semi-Asymmetric Graph Algorithms for NVRAMs. *Proceedings of the VLDB Endowment* (2020).

[9] Guanyu Feng, Zixuan Ma, Daixuan Li, Shengqi Chen, Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2021. Risgraph: A real-time streaming system for evolving graphs to support sub-millisecond per-update analysis at millions ops/s. In *Proceedings of the International Conference on Management of Data (SIGMOD)*.

[10] Yehonatan Fridman, Suprasad Mutalik Desai, Navneet Singh, Thomas Willhalm, and Gal Oren. 2023. CXL memory as persistent memory for disaggregated hpc: A practical approach. In *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*.

[11] Yehonatan Fridman, Yaniv Snir, Matan Rusanovsky, Kfir Zvi, Harel Levin, Danny Hendler, Hagit Attiya, and Gal Oren. 2021. Assessing the use cases of persistent memory in high-performance scientific computing. In *2021 IEEE/ACM 11th Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS)*. IEEE, 11–20.

[12] Chao Gao, Mahbod Afarin, Shafiur Rahman, Nael Abu-Ghazaleh, and Rajiv Gupta. 2023. Mega evolving graph accelerator. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[13] Sayan Ghosh, Nathan R Tallent, Marco Minutoli, Mahantesh Halappanavar, Ramesh Peri, and Ananth Kalyanaraman. 2021. Single-node partitioned-memory for huge graph analytics: cost and performance trade-offs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

[14] Gurbinder Gill, Roshan Dathathri, Loc Hoang, Ramesh Peri, and Keshav Pingali. 2020. Single machine graph analytics on massive datasets using Intel optane DC persistent memory. *Proceedings of the VLDB Endowment* (2020).

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

[16] Shenyang Huang, Farimah Poursafaei, Jacob Danovitch, Matthias Fey, Weihua Hu, Emanuele Rossi, Jure Leskovec, Michael Bronstein, Guillaume Rabusseau, and Reihaneh Rabbany. 2024. Temporal graph benchmark for machine learning on temporal graphs. *Advances in Neural Information Processing Systems* (2024).

[17] Yuriy Hulovatyy, Huili Chen, and Tijana Milenković. 2015. Exploring the structure and function of temporal networks with dynamic graphlets. *Bioinformatics* 31, 12 (2015), i171–i180.

[18] Intel. [n. d.]. Intel Optane Persistent Memory. https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/overview.html.

[19] Intel. 2022. Intel Reports Second-Quarter 2022 Financial Results. https://www.intc.com/news-events/press-releases/detail/1563/intel-reports-second-quarter-2022-financial-results.

[20] Intel. [n. d.]. Persistent Memory Development Kit. https://pmem.io/pmdk/.

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

[22] Abdullah Al Raqibul Islam and Dong Dai. 2023. DGAP: Efficient Dynamic Graph Analysis on Persistent Memory. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

[23] Abdullah Al Raqibul Islam, Dong Dai, and Dazhao Cheng. 2022. VCSR: Mutable CSR Graph Format Using Vertex-Centric Packed Memory Array. In *IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*.

[24] Anand Iyer, Li Erran Li, and Ion Stoica. 2015. CellIQ: Real-Time Cellular Network Analytics at Scale. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

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

[26] Joseph Izraelevitz, Hammurabi Mendes, and Michael L Scott. 2016. Linearizability of persistent memory objects under a full-system-crash failure model. In *International Symposium on Distributed Computing (DISC)*.

[27] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. 2019. Basic performance measurements of the intel optane DC persistent memory module. *arXiv preprint arXiv:1903.05714* (2019).

[28] Xiaoen Ju, Dan Williams, Hani Jamjoom, and Kang G Shin. 2016. Version traveler: Fast and memory-efficient version switching in graph processing systems. In *USENIX Annual Technical Conference (ATC)*.

[29] Myoungsoo Jung. 2022. Hello bytes, bye blocks: Pcie storage meets compute express link for memory expansion (CXL-SSD). In *Proc. ACM Workshop on Hot Topics in Storage and File Systems (HotStorage)*.

[30] Anuj Kalia, David Andersen, and Michael Kaminsky. 2020. Challenges and solutions for fast remote persistent memory access. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*.

[31] Wook-Hee Kim, R Madhava Krishnan, Xinwei Fu, Sanidhya Kashyap, and Changwoo Min. 2021. PACTree: A high performance persistent range index using PAC guidelines. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*.

[32] KIOXIA. 2022. Kioxia Launches Second Generation of High-Performance, Cost-Effective XL-FLASH Storage Class Memory Solution. https://www.kioxia.com/en-jp/business/news/2022/20220802-1.html.

[33] Dimitrios Koutsoukos, Raghav Bhartia, Michal Friedman, Ana Klimovic, and Gustavo Alonso. 2023. NVM: Is it Not Very Meaningful for Databases? *Poceedings of the VLDB Endowment* (2023).

[34] Pradeep Kumar and H Howie Huang. 2019. GRAPHONE: a data store for real-time analytics on evolving graphs. In *USENIX Conference on File and Storage Technologies (FAST)*.

[35] Jérôme Kunegis. 2013. Konect: the Koblenz Network Collection. In *Proc. International Conference on World Wide Web (WWW)*. 1343–1350.

[36] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media?. In *Proc. ACM World Wide Web (WWW)*.

[37] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-Scale graph computation on just a PC. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[38] Tianxi Li, Yang Wang, and Xiaoyi Lu. 2023. On the Discontinuation of Persistent Memory: Looking Back to Look Forward. In *Workshop on Hot Topics in System Infrastructure, Co-located with ISCA 2023*.

[39] 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 Sixteenth European Conference on Computer Systems (EuroSys)*.

[40] Soklong Lim, Zaixin Lu, Bin Ren, and Xuechen Zhang. 2019. Enforcing crash consistency of evolving network analytics in non-volatile main memory systems. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE.

[41] Peter Macko, Virendra J Marathe, Daniel W Margo, and Margo I Seltzer. 2015. LLAMA: Efficient Graph Analytics using Large Multiversioned Arrays. In *IEEE International Conference on Data Engineering (ICDE)*.

[42] Jasmina Malicevic, Subramanya Dulloor, Narayanan Sundaram, Nadathur Satish, Jeff Jackson, and Willy Zwaenepoel. 2015. Exploiting NVM in large-scale graph analytics. In *Proceedings of the 3rd Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*.

[43] Mugilan Mariappan and Keval Vora. 2019. Graphbolt: Dependency-driven synchronous processing of streaming graphs. In *Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys)*.

[44] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. 2013. Differential Dataflow. In *CIDR*.

[45] Youshan Miao, Wentao Han, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Enhong Chen, and Wenguang Chen. 2015. Immortalgraph: A system for storage and analysis of temporal graphs. *ACM Transactions on Storage (TOS)* (2015).

[46] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 439–455.

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

[48] Onkar Patil, Latchesar Ionkov, Jason Lee, Frank Mueller, and Michael Lang. 2019. Performance characterization of a dram-nvm hybrid memory architecture for hpc applications using intel optane dc persistent memory modules. In *Proceedings of the International Symposium on Memory Systems*.

[49] Arthur Sainio Pekon Gupta. 2020. Gen-Z emerging technology for memory intensive applications, In Storage Developer Conference (SDC), SNIA. https://www.snia.org/educational-library/gen-z-emerging-technology-memory-intensive-applications-2020.

[50] Ivy B Peng, Maya B Gokhale, and Eric W Green. 2019. System evaluation of the intel optane byte-addressable nvm. In *Proceedings of the International Symposium on Memory Systems*. 304–315.

[51] Chenghui Ren, Eric Lo, Ben Kao, Xinjie Zhu, and Reynold Cheng. 2011. On querying historical evolving graph sequences. *Proceedings of the VLDB Endowment* (2011).

[52] Ryan Rossi and Nesreen Ahmed. 2015. The network data repository with interactive graph analytics and visualization. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 29.

[53] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*.

[54] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M Tamer Özsu. 2017. The ubiquity of large graphs and surprising challenges of graph processing. *Proceedings of the VLDB Endowment* (2017).

[55] Samsung. [n. d.]. Samsung CXL Solutions – CMM-H . https://semiconductor.samsung.com/us/news-events/tech-blog/samsung-cxl-solutions-cmm-h/.

[56] Samsung. 2022. Memory-Semantic SSDTM: Industry 1st CXL-Based Storage Optimized for AI/ML. https://samsungmsl.com/ms-ssd/.

[57] Shintaro Sano, Yosuke Bando, Kazuhiro Hiwada, Hirotsugu Kajihara, Tomoya Suzuki, Yu Nakanishi, Daisuke Taki, Akiyuki Kaneko, and Tatsuo Shiozawa. 2023. GPU graph processing on cxl-based microsecond-latency external memory. In *Proceedings of the SC'23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis*. 962–972.

[58] Korakit Seemakhupt, Sihang Liu, Yasas Senevirathne, Muhammad Shahbaz, and Samira Khan. 2021. PMNet: In-network data persistence. In *ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*.

[59] Kapil Sethi. [n. d.]. Expanding the Limits of Memory Bandwidth and Density: Samsung's CXL Memory Expander. https://semiconductor.samsung.com/news-events/tech-blog/expanding-the-limits-of-memory-bandwidth-and-density-samsungs-cxl-dram-memory-expander/.

[60] 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 International Conference on Management of Data (SIGMOD)*.

[61] Julian Shun and Guy E Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*.

[62] Keval Vora, Rajiv Gupta, and Guoqing Xu. 2017. Kickstarter: Fast and Accurate Computations on Streaming Graphs via Trimmed Approximations. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[63] Rui Wang, Shuibing He, Weixu Zong, Yongkun Li, and Yinlong Xu. 2022. XPGraph: XPline-friendly Persistent Memory Graph Stores for Large-scale Evolving Graphs. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[64] Michèle Weiland, Holger Brunst, Tiago Quintino, Nick Johnson, Olivier Iffrig, Simon Smart, Christian Herold, Antonino Bonanni, Adrian Jackson, and Mark Parsons. 2019. An early evaluation of intel's optane dc persistent memory module and its impact on high-performance scientific applications. In *Proceedings of the international conference for high performance computing, networking, storage and analysis (SC)*.

[65] Brian Wheatman and Helen Xu. 2018. Packed compressed sparse row: A dynamic graph representation. In *IEEE High Performance Extreme Computing Conference (HPEC)*.

[66] Brian Wheatman and Helen Xu. 2021. A parallel packed memory array to store dynamic graphs. In *2021 Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 31–45.

[67] Mincheng Wu, Chao Li, Zhangchong Shen, Shibo He, Lingling Tang, Jie Zheng, Yi Fang, Kehan Li, Yanggang Cheng, Zhiguo Shi, et al. 2022. Use of temporal contact graphs to understand the evolution of COVID-19 through contact tracing data. *Communications Physics* 5, 1 (2022), 270.

[68] Lingfeng Xiang, Xingsheng Zhao, Jia Rao, Song Jiang, and Hong Jiang. 2022. Characterizing the performance of intel optane persistent memory: A close look at its on-dimm buffering. In *Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys)*.

[69] 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 *USENIX Conference on File and Storage Technologies (FAST)*.

[70] Shao-Peng Yang, Minjae Kim, Sanghyun Nam, Juhyung Park, Jin-Yong Choi, Eyee Hyun Nam, Eunji Lee, Sungjin Lee, and Bryan S Kim. 2023. Overcoming the Memory Wall with {CXL-Enabled}{SSDs}. In *USENIX Annual Technical Conference (ATC)*.

[71] Xizhe Yin, Zhijia Zhao, and Rajiv Gupta. 2024. IncBoost: Scaling Incremental Graph Processing for Edge Deletions and Weight Updates. In *Proceedings of the 2024 ACM Symposium on Cloud Computing*.

[72] Wenhui Zhang, Xingsheng Zhao, Song Jiang, and Hong Jiang. 2021. Chameleondb: a key-value store for optane persistent memory. In *Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys)*.

[73] Xu Zhang, Yisong Chang, Tianyue Lu, Ke Zhang, and Mingyu Chen. 2023. Rethinking design paradigm of graph processing system with a cxl-like memory semantic fabric. In *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 25–35.