DGAP: Efficient Dynamic Graph Analysis on Persistent Memory

Abdullah Al Raqibul Islam
Computer Science Department,
University of North Carolina at Charlotte
Charlotte, NC, USA
aislam6@uncc.edu

Dong Dai
Computer Science Department,
University of North Carolina at Charlotte
Charlotte, NC, USA
ddai@uncc.edu

ABSTRACT

Dynamic graphs, featuring continuously updated vertices and edges, have grown in importance for numerous real-world applications. To accommodate this, graph frameworks, particularly their internal data structures, must support both persistent graph updates and rapid graph analysis simultaneously, leading to complex designs to orchestrate 'fast but volatile' and 'persistent but slow' storage devices. Emerging persistent memory technologies, such as Optane DCPMM, offer a promising alternative to simplify the designs by providing data persistence, low latency, and high IOPS together. In light of this, we propose DGAP, a framework for efficient dynamic graph analysis on persistent memory. Unlike traditional dynamic graph frameworks, which combine multiple graph data structures (e.g., edge list or adjacency list) to achieve the required performance, DGAP utilizes a single mutable Compressed Sparse Row (CSR) graph structure with new designs for persistent memory to construct the framework. Specifically, DGAP introduces a *per-section edge log* to reduce write amplification on persistent memory; a *per-thread undo log* to enable high-performance, crash-consistent rebalancing operations; and a data placement schema to minimize in-place updates on persistent memory. Our extensive evaluation results demonstrate that DGAP can achieve up to 3.2× better graph update performance and up to 3.77× better graph analysis performance compared to state-of-the-art dynamic graph frameworks for persistent memory, such as XPGraph, LLAMA, and GraphOne.

ACM Reference Format:
Abdullah Al Raqibul Islam and Dong Dai. 2023. DGAP: Efficient Dynamic Graph Analysis on Persistent Memory. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '23), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3581784.3607106

1 INTRODUCTION

The ability to ingest new graph data continuously and analyze the latest graphs efficiently is crucial for many real-world applications today. For instance, cellular network operators need to address traffic hotspots in their networks as they are generated and identified [27]. A dynamic graph framework that can both *persistently store new graph updates* and *perform complex graph analysis on the latest graph* is essential for supporting such applications. However, constructing such a framework is fundamentally challenging. Existing storage devices like SSDs, hard disks, or DRAM either lack persistence (as in volatile DRAM) or offer low performance on graph analysis (like SSDs and hard disks). To handle both operations, graph frameworks must manage various storage devices, design unique data structures for each, and find a balance between them, leading to intricate systems. For example, GraphOne persists the graphs updates on SSD in Edge List (EL), conducts graph analysis on DRAM using Adjacency List (AL), and continuously synchronizes data between the two [33].

Recently, a new set of non-volatile or persistent memory devices (PMs) have emerged, such as Intel Optane DC Persistent Memory [46]. These devices can be accessed in bytes via the memory bus with data persistence guarantees. Compared to DRAM, PMs offer data persistence and greater density (e.g., Optane's 512GB/dimm vs. DRAM's 64GB/dimm). Compared to block-based devices, PMs allow byte-level access using `load` and `store` instructions with significantly lower latency (e.g., ~300 ns vs. ~100 ms) and higher IOPS (e.g., ~10M vs. ~500K for random writes) [51, 52, 61, 72]. These characteristics suggest a promising alternative for building dynamic graph frameworks: *employ PMs to serve both graph updates and graph analysis* for persistence, speed, and capacity. This approach further avoids the cost of data movements and reduces the complexity of coordinating multiple data structures on different storage devices. Although Intel has discontinued Optane PMs due to business reasons, millions of these devices remain available, and various new non-volatile memory solutions continue to emerge. We contend that designing high-performance storage systems on persistent memory devices remains both economically practical and beneficial, as evidenced by recent studies [60, 64].

However, directly porting existing graph frameworks to PMs can be sub-optimal. Existing dynamic graph frameworks, such as LLAMA [42] or GraphOne [33], utilize block I/O interfaces, whose software overheads are not acceptable for byte-addressable PMs [68]. The data structures are not tailored for PMs either, leading to potential performance issues [25, 26]. Moreover, although PMs are persistent devices, writing data persistently is complicated due to the existence of volatile CPU caches. Extra flushing and fencing operations, though necessary, become costly without the right optimizations [25, 26]. Unexpected crashes further necessitates expensive transactions to avoid partial writes, significantly impacting the performance [21, 67].

On the other hand, existing PM-specific dynamic graph frameworks, such as NVGRAPH [40] and, more recently, XPGraph [64], continue to follow the traditional approach of coordinating separate persistence-friendly and analysis-friendly data structures (i.e., edge list or adjacency list) on DRAM or PMs. This approach still leads to

SC '23, November 12–17, 2023, Denver, CO, USA

ACM ISBN 979-8-4007-0109-2/23/11…$15.00
https://doi.org/10.1145/3581784.3607106

overly complicated data synchronization between data structures and creates unnecessary conversions or movements.

In this study, we introduce a novel approach to design a unified graph data structure, serving both graph persistence and analysis directly from persistent memory. To this end, we propose DGAP, a Dynamic Graph Analysis framework specifically designed for Persistent memory. DGAP is built upon a recently proposed mutable Compressed Sparse Row (CSR) graph structure [24, 66], which leverages Packed Memory Array (PMA) [5] for efficient graph updates and analysis. Instead of naively porting mutable CSR to PMs, DGAP introduces a series of new designs to enhance its performance on PMs. Firstly, DGAP introduces a new *per-section edge log* data structure to mitigate the write amplification issues associated with mutable CSR. Secondly, DGAP integrates new *per-thread undo logs* to support high-performance crash-consistent rebalancing operations, which are frequent and costly operations in mutable CSR. Thirdly, DGAP strategically caches various mutable CSR components in DRAM according to the workloads. Through these designs, DGAP is able to deliver exceptional performance on both graph updates and graph analysis by maximally utilizing PMs.

We implemented DGAP in around 2,000 lines of C++ code and compared its performance to that of state-of-the-art graph frameworks on PMs, using multiple graph analysis algorithms on different real-world graphs. Our results show that DGAP achieves up to 3.2× improved graph update performance and 3.77× enhanced graph analysis performance compared to leading graph frameworks, such as XPGraph, LLAMA, and GraphOne.

The remainder of this paper is organized as follows: In §2 we discuss the background and motivation of this study. We introduce persistent memory device, existing graph storage formats including PMA-based mutable CSR, and most importantly, why directly porting PMA-based mutable CSR to PMs does not work. In §3, we present the key components of DGAP and its operations in details. We present the extensive experimental results in §4. We compare with related work in §5, conclude this paper and discuss the future work in §6.

2 BACKGROUND AND MOTIVATION

2.1 PMs and Optane DCPMM Overview

2.1.1 Overview. Persistent memory describes storage devices that are accessible in bytes via memory interfaces and can retain the stored data after the power is off [1, 32, 36, 53]. Intel Optane DC Persistent Memory is the first commercially available PMs [46, 62]. Working on Intel Cascade Lake platforms, Optane can scale up to 24TB in a single machine [13]. It can be configured in either *Memory* mode or *App Direct* mode [50]. In *Memory* mode, the Optane devices are exposed as DRAM, with the actual DRAM becomes a transparent 'L4' cache to accelerate data access. However, this model does not support data persistence. In *App Direct* mode, Optane devices are directly exposed to users alongside DRAM. This mode allows users to access both DCPMM and DRAM and offers data persistence capability. In this study, we focus on *App Direct* mode.

2.1.2 Performance Features. PMs exhibit performance characteristics critical for building graph storage on them. For instance, their writes are slower due to the added persistence cost. The performances of large sequential accesses are often better than small random accesses due to the internal read/write buffers in these devices. Here, we use Optane DCPMM as an example to further highlight some performance features [25, 26, 28, 59, 62, 69, 70]. Firstly, the read/write performance of Optane DCPMM is asymmetric. Write operations, particularly persistent ones, incur significant overheads (e.g., up to ~ 7-8× slower than DRAM). In contrast, read latencies are around ~ 2-3× slower than DRAM. This underscores the importance of minimizing unnecessary writes. Secondly, since Optane DCPMM uses 256 bytes internal write buffers, small random writes will perform much worse than large sequential writes. It is then critical to ensure the writes can be properly grouped [20].

2.1.3 Persistence Features. The challenge to achieve persistence in PMs is that not all the components in the memory hierarchy is persistent. Optane DCPMM introduces a concept called Asynchronous DRAM Refresh (ADR) which ensures during a power loss, all data in ADR will be written to PMs. But ADR does not include CPU caches. To guarantee data persistence, programmers must explicitly call `CLFLUSHOPT` and `SFENCE` instructions to flush the cache line and enforce the memory operations order [9]. But even with the cache line flushed and memory fenced, large writes to PMs may still be partially persisted as its atomic write unit is small (i.e., 8 bytes). Transactions are essential for ensuring data safety during large writes, yet they can significantly affect the performance, as recent research suggested [21, 67]. Lately, extended ADR (eADR) was introduced in the 3rd generation Intel Xeon Scalable Processors to make CPU caches included in the power fail protected domain [14]. The eADR feature greatly simplified the programming [73]. But it is not available in all PMs platforms. The applications need to recognize the devices and perform correctly and efficiently regardless which platforms are supported. DGAP is implemented to work with both ADR and eADR platforms.

2.2 Graph Store and CSR

At the heart of graph frameworks are their storage data structures. There have been a significant number of graph storage data structures, such as edge list (EL), adjacency list (AL), Compressed Sparse Row (CSR), and many others [7, 55] used in different graph frameworks [16, 33, 34, 42, 54, 57, 75].

EdgeList (EL) is a sequential edge log, efficient for edge additions but slow for vertex accesses since it requires scanning the entire edge log. The Adjacency List (AL) and its variations, like blocked adjacency list [49], use a per-vertex linked list for storing vertex neighbors. While perform well at graph insertions and single vertex operations, they struggle with whole graph analysis due to memory overheads and cache inefficiencies [33, 42].

Compressed sparse row (CSR), on the other hand, is designed for efficient graph analysis. It groups all edges from the same vertex together and stores them sequentially in an edge array, while the vertex array stores each vertex's starting index. In this way, CSR supports both per-vertex queries and edge iterations efficiently. It delivers extreme graph analysis performance because most of the vertices and edges are accessed sequentially. Its major limitation, however, is that it can not accommodate dynamic graph updates without rebuilding the entire edge array for each edge

insertion. To address this limitation, recent studies have proposed to use the Packed Memory Array (PMA) to make the edge array mutable [15, 24, 56, 65]. Such mutable CSR data structures can offer extreme graph analysis performance while handling graph updates efficiently, making them a perfect candidate to build the PMs-based graph framework.

2.3 PMA-based Mutable CSR

The Packed Memory Array (PMA) is fundamentally a sorted array with reserved empty gaps interspersed [5]. These gaps provide room for future insertions without shifting the entire array. To maintain the gap density, PMA employs a binary *PMA Tree* to track density changes in different sections of the array. For any section located at tree height i, PMA assigns the lower and upper bound density thresholds as ρ_i and τ_i. When insertions or deletions make the density of a section out of the range, PMA initiates *rebalancing* operations to adjust its gap density by redistributing gaps among adjacent sections. The rebalancing will happen at a level where all affected sections' densities together will fall within the density range. If the whole array is full, PMA *resizes* the array by increasing its size. The amortized write overhead for adaptive PMA is $O(\log N)$. More details about PMA can be found in [5].

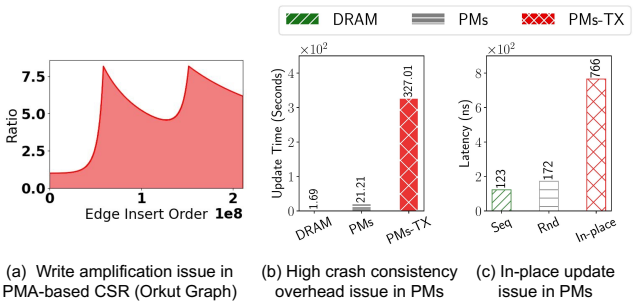
PMA-based mutable CSRs incorporate this concept by replacing the original CSR edge array with the packed memory array, exemplified by PCSR [65] first. VCSR [24] further optimizes PCSR by considering the skewed workloads inherent in real-world graphs. It partitioned the edge array into varied-size sections and distributed the gaps unevenly based on historical workloads in each section to improve performance.

2.4 Issues of Mutable CSR on PMs

PMA-based mutable CSR has been proven effective to support both graph updates and analysis. However, due to the unique features of PMs, a naive implementation leads to problematic performance, as summarized in the later three issues.

2.4.1 Write Amplification Issue. Although mutable CSR avoids shifting the entire edge array for insertion, it still requires shifting a small range of elements if the targeted insertion location is occupied. These additional shifts result in write amplifications. Compared to DRAM, write amplifications in PMs are more critical due to PMs' asymmetric read/write performance. Additionally, these *nearby shifts* often occur within a range smaller than 256 bytes, the size of the Optane DCPMM internal write buffer. This forces the buffers to be flushed before being filled, leading to inefficient buffer utilization. To illustrate the issue, we inserted the real-world graph, *Orkut* [38], into a mutable CSR implementation and calculated the ratio of actual memory writes v.s the edge size (write amplification) during insertions. Figure 1(a) reported the ratio during insertions. We can observe that the write amplification can be as high as 7×. It is hence critical to address such an issue.

2.4.2 Crash Consistency Issue. In addition to *nearby shifts*, insertions could further trigger PMA *rebalancing* when a section becomes full. These rebalancing operations move large chunks of sequential elements to new locations. Although efficient in DRAM, these operations are costly on PMs due to the persistence guarantee.



(a) Write amplification issue in PMA-based CSR (Orkut Graph) (b) High crash consistency overhead issue in PMs (c) In-place update issue in PMs

Figure 1: Issues of PMA-based mutable CSRs on PMs. ***The evaluation platform is described in Sec. §4.1.***

It is necessary to use transactions to protect large chunks of writes. However, as demonstrated in Figure 1(b), transactions are extremely expensive on PMs. The time required to insert a graph into DRAM, PMs (without transactions), and PMs-TX (with transactions) differ substantially [25, 26]. Therefore, it is crucial to develop efficient crash recovery for frequent *rebalancing* operations.

2.4.3 In-place Update Issue. In-place updates in DRAM are efficient, leveraging the cache. But, persistent in-place updates on PMs are exactly the opposite. Figure 1(c) illustrates the performance of in-place updates on PMs. We present the latency of writing the same size of data in a sequential (Seq), random (Rnd), and in-place (In-place) manner respectively. We can observe 7× difference in latency. The reason is that persistent in-place updates repeatedly flush the same cache line and dramatically slow down the performance due to the blocking of previous flushing operations and possible on-chip wear-leveling protection [28]. Crucial components of mutable CSR, such as the vertex degree and the PMA tree, require frequent in-place updates. Conducting these updates directly on PMs would be significantly slow. It is essential to design the data placement strategy to minimize in-place updates on PMs.

3 DGAP DESIGN AND IMPLEMENTATION

DGAP, as illustrated in Fig.2, is designed to address the three issues outlined in Sec. §2.4. Its architecture comprises four primary components: ❶ *vertex array*, ❷ *edge array*, ❸ *per-section edge log*, and ❹ *per-thread undo log*. When interacting with DGAP, users launch multiple *writer threads* for graph updates and can execute multi-thread graph analysis *tasks* on the latest graphs. DGAP ensures the analysis tasks access only the latest graph snapshot when they start. This guarantees the long-running multi-iteration graph algorithms can access a consistent graph throughout their run.

❶ *Vertex Array.* DGAP stores all vertices sequentially in the vertex array. These sequential vertex IDs result from pre-processing by upstream applications, and their range is often known. Consequently, DGAP can pre-allocate the vertex array accordingly. Each vertex (v) in the vertex array takes 16 bytes to store three key pieces of metadata: its current degree ($degree_v$, 4 Bytes), starting index in the edge array ($start_v$, 8 Bytes), and a pointer to its *per-section edge log* (el_v, 4 Bytes). The most important design decision about the DGAP vertex array is placing it entirely in DRAM. The main reason behind this design decision is to prevent frequent in-place updates

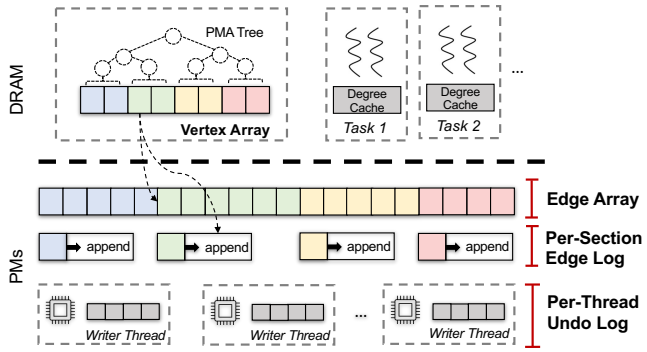


Figure 2: Overall architecture of DGAP.

on PMs. For dynamic graphs, the vertex degree ($degree_v$) must be updated each time an edge is inserted. The pointer to the *per-section edge log* (el_v) also changes when edges are added to the edge log. Both operations are frequent enough to significantly impact overall performance if executed as in-place updates on PMs. Storing them in DRAM effectively avoids this issue.

Data safety is a critical issue when storing the entire vertex array in DRAM. DGAP introduces a new pivot element for each vertex in the edge array and leverages these elements to reconstruct the entire vertex array after a crash. More details are provided in Sec. §3.1.5. The reconstruction is fast due to the high bandwidth of PMs for sequential accesses. Detailed results are reported in the evaluation section. Another potential concern is DRAM capacity. Theoretically, each DGAP vertex takes 16 bytes, so 16GB DRAM can store 1 billion vertices. Since most graphs have more edges than vertices, we anticipate that the capacity issue will primarily affect the PMs edge array rather than the DRAM vertex array.

❷ *Edge Array.* DGAP stores all the edges in the edge array on persistent memory. The edge array is a PMA constructed based on the VCSR strategy [24]. Following other dynamic graph frameworks (e.g., XPGraph, GraphOne), each DGAP edge takes 4 bytes as it only stores the destination vertex ID. Storing the source vertex ID is unnecessary, as it is shared by all edges originating from the same vertex. The source vertex ID is instead stored as a pivot element at the beginning of each vertex' edge list. The pivot element serves as additional metadata in DGAP to reconstruct the DRAM vertex array after crashes. Specifically, the pivot is a special 'edge' element with a value of *–vertex-id*. Since it is negative and illegal as a vertex ID, it can be used to denote the start of the vertex during recovery. Further details about DGAP recovery are in Sec. §3.1.5.

One important design decision regarding the DGAP edge array is the storage order of all edges for a vertex. Traditionally, the edges of a vertex are sorted based on their destination vertex ID [65]. However, DGAP stores them according to their insertion order, meaning a new edge will always be stored at the end of the vertex' edge list. So, an edge $(1 \rightarrow 2)$ may be stored after edge $(1 \rightarrow 6)$. This seemingly minor change is critical for DGAP to maintain a consistent snapshot of the latest graph for analysis tasks. This means that for any vertex v, if we know its degree at time t ($degree_v^t$), we can easily determine its readable edges for $Task_t$, which should fall within the range $[start_v, start_v + degree_v^t)$. Any edge after that will not be visible to $task_t$. Hence, creating a snapshot of the latest graph only involves storing the degrees of all vertices at time t. At present, we simply cache this degree info in the *Degree Cache* within each task's DRAM space, as shown in Fig. 2. This can be done at the beginning of the analysis tasks. The primary issue here is memory cost. Many of the degrees are the same and do not need to be stored in each task. In the future, we plan to implement a Copy-on-Write (CoW) Degree Cache so that all tasks and the main vertex array can share unchanged degrees without wasting memory.

❸ *Per-section Edge Log.* A primary performance challenge in existing PMA-based mutable CSRs on PMs is the write amplifications caused by *nearby shifts* within each PMA section during insertions. To mitigate this, our principal approach is to temporarily hold these insertions in a persistent log and merge them back in batches later. We introduce the concept of *per-section edge logs* in DGAP, representing a pre-allocated, continuous, fixed-size space (`ELOG_SZ`) on PMs dedicated for each PMA section. These logs temporarily store new edge insertions when a *nearby shift* becomes necessary. In our prototype, `ELOG_SZ` is set to 2K bytes.

Each element stored in the *edge log* contains three metadata components and occupies 12 bytes: (i) source vertex ID, (ii) destination vertex ID, and (iii) a back-pointer. This back-pointer is designed to connect all edges originating from the same source vertex, arranging them in reverse order within the edge log. The most recent edge points back to the preceding edge of the same source vertex in the log. The edge log pointer (el_v), stored in the *vertex array*, pinpoints the most recent edge of a vertex in the edge log. A detailed insertion workflow of DGAP is shown in Fig. 3.

When the *per-section edge log* reaches 90% usage, a *merging* operation is initiated, integrating the edge log data back into the edge array. Notably, edges within the *edge log* also contribute to the density of the corresponding *edge array* section. Therefore, the standard PMA rebalancing operations might be triggered if either the edge array or edge log is approaching full capacity. During DGAP rebalancing, data from both the edge array sections and their respective edge logs are considered.

❹ *Per-thread Undo Log.* PMA *Rebalancing*, which redistributes gaps among sections, is critical for mutable CSR. To ensure data safety, it requires transaction mechanisms to avoid partial writes and guarantee crash consistency. While existing PMs programming libraries like PMDK [50] support transactions natively, using them directly for recurrent *rebalancing* operations results in significant overhead, due to two major bottlenecks: 1) the high memory allocation cost of frequent journal allocations and 2) performance overheads due to excessive ordering [21]. In DGAP, we introduce a *per-thread undo log* specifically to enhance the performance of *rebalancing* while ensuring crash consistency. During insertion, whenever a *Writer Thread* triggers rebalancing, before actually moving data, it first uses its own *undo log* to persistently backup the data set to be relocated, chunk by chunk, acting as an 'undo log'. If a crash happens in the middle, we can recover the data from the undo log. The *per-thread undo log* is pre-allocated in fixed size (i.e., `ULOG_SZ`) for each *Writer Thread*. In our prototype, `ULOG_SZ` is set to 2K bytes.

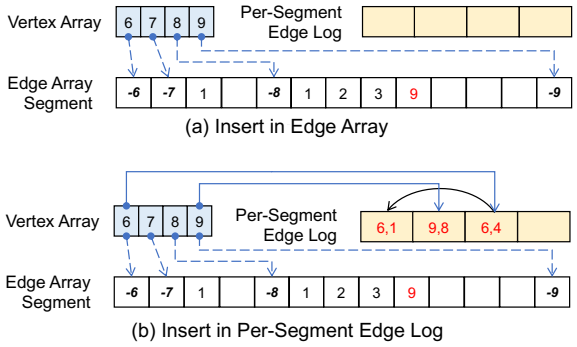


Figure 3: Two insertion cases in DGAP. ***The dashed blue line points to the starting index of a vertex in the edge array.***

3.1 DGAP Graph Operations

This section explains how the DGAP components work together to serve various graph operations.

3.1.1 Initialization. When DGAP starts for the first time, it takes multiple user-specific parameters for system initialization. The number of vertices and edges in the graph are specified by the parameters `INIT_VERTICES_SIZE` and `INIT_EDGES_SIZE`. DGAP allocates the initial vertex array in DRAM and the edge array in PMs accordingly. Both parameters are just initial user estimations. The actual numbers of vertices or edges can significantly surpass these values. When this happens, DGAP automatically resizes both the vertex and edge arrays during insertions. DGAP also utilizes the parameters `ELOG_SIZE` and `ULOG_SIZE` to pre-allocate the per-section edge logs and per-thread undo logs. Furthermore, DGAP initializes multiple key metadata pieces on PMs for its operation. For instance, it maintains a global flag, `NORMAL_SHUTDOWN`, on PMs to determine if DGAP had a graceful shutdown in its previous session. Whenever DGAP restarts, this value guides the system initialization process. In addition, DGAP creates and upholds various DRAM indexing metadata, including the PMA tree for density tracking. Locks are allocated based on this PMA tree to ensure concurrent reads/writes in DGAP. More details are discussed in later subsections.

3.1.2 Graph Updates. DGAP utilizes the PMA-based mutable CSR structure to enable dynamic graph updates. For edge updates, an edge pair (v_{src}, v_{dst}) will be fed to the `g.insertE()` call. For vertex updates, a vertex ID (v_{src}) will be fed to the `g.insertV()` call. Edge updates include both edge insertions and deletions. Deletions are executed by re-inserting the same edge marked with a tombstone flag. Specifically, we set the first bit of the destination vertex ID to 1, signifying that the edge has been removed from the graph. In the following, we delve into edge insertion operations.

Edge insertion includes two steps: 1) inserting the new edge into the *edge array* or *edge log*, and 2) updating the degree and pointer in the *vertex array*. The DRAM vertex array is updated only after the PMs edge array has been successfully updated and flushed. In this way, even crash happens after PMs updates, the DRAM data structures can be reconstructed afterward. Given that we store all edges of a vertex chronologically, the insertion point for a new edge $[v_{src}, v_{dst}]$ in the *edge array* can be easily determined. It can be calculated directly from the degree of v_{src} and its starting index using the formula $(start_{v_{src}} + degree_{v_{src}})$. If the calculated location is a gap, the new edge can be inserted in an atomic manner. However, if the location is taken by a subsequent vertex, which requires a *nearby shift*, DGAP appends the edge to the *per-section edge log* to minimize write amplification.

Fig. 3 illustrates two DGAP insertion scenarios. This figure provides a snapshot of the *vertex array*, *edge array*, and the associated *per-section edge log*. Here, edges for vertices (6, 7, 8, 9) are showcased on the edge array with gaps, while the *per-section edge log* is empty. Fig. 3(a) first shows a normal insertion case ($8 \rightarrow 9$) where the intended edge location is empty. Then the edge is inserted on the *edge array* (marked in red). Fig. 3(b) shows another scenario where the desired locations for a series of edge insertions (e.g., $6 \rightarrow 1$, $6 \rightarrow 4$) are already taken (by vertex 7 and its edges). In this case, new edges will be stored on the *per-section edge log* to reduce the unnecessary data shifts within the *edge array*. Multiple edges of the same vertex on the edge log will be connected using the back-pointer, shown as the black arrow from (6, 4) to (6, 1) in Fig. 3(b).

After many edge insertions, the corresponding section of the *edge array* is becoming full. This will trigger a PMA rebalancing operation that redistributes the gaps among adjacent sections to ensure all the sections maintain a satisfactory density. While DGAP adopts the same logic to initiate the rebalancing, it carries out the operation with assistance from the *per-thread undo log* to guarantee data consistency. Further details about crash-consistent rebalancing are elaborated in Sec 3.1.4.

3.1.3 Graph Analysis. DGAP supports graph analysis by offering high-performance interfaces to iterate through all vertices (i.e., `g.v()`) and the edges associated with a vertex (i.e., `v.e()`). Graph analysis tasks might run for extended durations. For instance, the PageRank algorithm executed on the *Orkut* graph can take over 20 seconds. During this time, the graph may be updated. To ensure a consistent view of the graph, it is necessary to guarantee that future reads from the same task bypass the newly added data. To achieve this, users must first call the `g.consistent_view()` function prior to iterating through the graph in their analysis tasks. Once this function is invoked, DGAP allocates a Degree Cache for the analysis task and temporarily holds the graph updates. It then copies the degree part of the vertex array to the per-task Degree Cache. This snapshot of degree information aids in pinpointing the appropriate set of edges for reading during task execution.

Once the snapshot is created, DGAP starts serving data-accessing function calls. For each call, DGAP initially reads the required metadata about v from DRAM *vertex array*, then accesses the PMs edge array based on that. The necessary metadata from vertex array includes the starting index $(start_v)$ and the edge log pointer (el_v). The degree information is obtained from the Degree Cache created at the task starting time t $(degree_v^t)$. If el_v is `NULL`, then iterating through v's edges involves simply iterating the corresponding *edge array* from $start_v$ to $(start_v + degree_v^t)$. If el_v is not `NULL`, the edges also come from the edge log. In this case, we first scan the edge array. If the edge array does not contain a sufficient number of edges as needed (based on $degree_v^t$), DGAP proceeds to scan the edge log. The el_v pointer always points to the last edge. From this point, we track all edges in the edge log through their back-pointer.

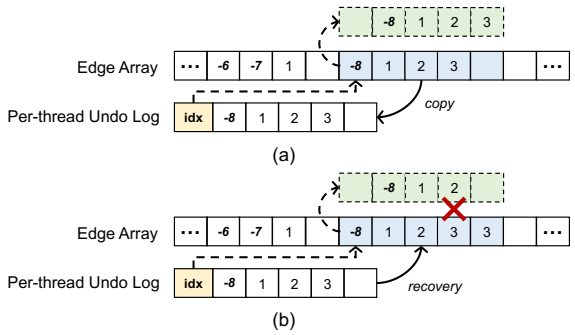


Figure 4: Crash consistent PMA *rebalancing* in DGAP. In (a), the blue area shows the intended data movement region; the green dashed boxes show the expected state after the data movement. In (b), a crash case is shown after moving data 3.

To read only the required number of edges (assuming $rest_v^t$), we allocate a first-come-first-out (FIFO) buffer with a size of $rest_v^t$ to keep only the necessary edges.

3.1.4 Crash Consistent PMA Rebalancing. Thus far, we have discussed how DGAP handles point insertions without initiating rebalancing operations. In PMA, however, rebalancing is crucial when the insertions of new edges makes sections of the edge array overly dense. These rebalancing operations redistribute gaps among neighboring sections to alleviate the density issue. Given that rebalancing involves considerable amount of data movement, it necessitates crash-consistent transactions. Yet, standard PMDK transactions are proven to be overly expensive. As a solution, DGAP introduces *per-thread undo logs* to achieve more efficient, crash-consistent DGAP rebalancing.

For every *Writer Thread* in DGAP, a *per-thread undo log* is allocated on PMs to support the execution of triggered *rebalancing*. Fig. 4 illustrates the rebalancing process in detail. Once DGAP determines a valid rebalancing range based on density thresholds, it recalculates the location of each vertex within these sections, assuming the gaps will be redistributed evenly. For instance, in Fig. 4(a), the new location of vertex v_8 and its edges is represented by dashed boxes above the edge array. During rebalancing, all vertices and their edges must be moved to their new locations. To prevent permanent data loss in the event of a crash, this relocation process must be safeguarded using a transaction mechanism.

To perform data movements in a crash-consistent manner, DGAP first backs up the data that may be overwritten during data movements in the *undo log*. It then calls `CLWB` and `SFENCE` to ensure that the data is persisted before proceeding with the actual data movement on the edge array. If a crash occurs before the backup of data on the undo log is completed, the data on the original edge array remains unaffected, as no data movement has occurred yet. After the backup, DGAP initiates the process of moving and overwriting data element by element. DGAP iteratively performs these steps until the entire rebalancing range is moved. In each step, it moves a maximum of `ULOG_SZ`=2K bytes of data.

Figure 4(b) illustrates a crash scenario during rebalancing. In this instance, DGAP has already backed up the moving data in the *undo log* and is beginning to shift all edges of v_8 one element to the right. Suppose that after the edge (8, 3) has been moved, a crash occurs, resulting in an inconsistent *edge array* due to the presence of two edges (8, 3). However, a consistent backup of this region is available in the persistent *undo log*. Upon restart, DGAP recognizes the crash by checking its `NORMAL_SHUTDOWN` flag. It then iterates through all *per-thread undo logs* and utilizes the backup data to overwrite the inconsistent regions. The *idx* index, stored at the beginning of the *per-thread undo log*, is used to determine which part of the edge array should be overwritten for recovery. After restoring the data, DGAP proceeds to reissue the rebalancing operation to complete the interrupted process.

3.1.5 Shutdown and Crash Recovery. DGAP can initiate a graceful shutdown by calling `g.shutdown()`. During a normal shutdown, DGAP first waits for all ongoing graph analytic tasks to complete. Subsequently, it persists all DRAM components to persistent memory (PM), including the *vertex array* and PMA-related metadata. While this backup process may require a few seconds, it ensures a quicker subsequent startup. Detailed normal shutdown times are measured and presented in the evaluation section. Before shutting down, DGAP resets the `NORMAL_SHUTDOWN` flag to indicate a graceful shutdown.

After rebooting, DGAP first checks the `NORMAL_SHUTDOWN` flag to understand whether the previous shutdown is normal or due to a crash. If the flag indicates a normal shutdown, DGAP simply loads the *vertex array* and PMA-related metadata to DRAM and starts operating. If this is a reboot after a crash, DGAP initiates a data recovery process. Initially, DGAP scans the *edge array* to reconstruct the *vertex array* and build PMA metadata, such as the density tree. Following that, DGAP scrutinizes all *per-thread undo logs* and recovers the inconsistencies resulting from crashed rebalancing operations. It then continues to finish the ongoing rebalancing from the inconsistent region. Next, DGAP checks the *per-section edge log* to retrieve the metadata for these vertices and update the *vertex array*. After all these steps, DGAP can start to operate normally. In the evaluation section, we present the time durations associated with both standard and crash reboots.

3.1.6 Concurrency Control. DGAP supports multi-thread graph updates (multiple *Writer Threads*) and graph analysis (multiple *Analysis Tasks*) on PMs. To optimize performance, DGAP implements an optimistic read/write lock to enable multiple readers and writers to run concurrently, as long as they do not write to the same section. For each PMA section, DGAP maintains a lock and its linked condition variable, resulting in $|log(v)|$ locks. When inserting an edge (v_{src}, v_{dst}), DGAP first needs to acquire the lock for the respective section of v_{src} so that no other threads can insert into the same section. This also prevents concurrent readers. After the insertion, DGAP checks whether the density of $Section_{v_{src}}$ has reached the rebalancing threshold. If rebalancing is needed, the writer thread first sets the condition variable of $Section_{v_{src}}$ to block other writes or rebalancing operations in this section. It then attempts to acquire all the locks of the sections affected by the rebalancing, sequentially. To prevent deadlocks, DGAP follows a strict order (from low to high section IDs) when acquiring locks. After obtaining all the locks, DGAP executes the rebalancing as previously described. Finally, DGAP resets the condition variable and notifies all waiting writes

Table 1: A list of graph kernels and inputs and outputs used in our evaluations.

Graph kernel	Kernel Type	Input	Output	Notes
PageRank (PR)	Link Analysis	-	$\|V\|$-sized array of ranks	Fixed number (20) of iterations
Breadth-First Search (BFS)	Graph Traversal	Source vertex	$\|V\|$-sized array of parent IDs	Direction-Optimizing approach [4]
Betweenness Centrality (BC)	Shortest Path	Source vertex	$\|V\|$-sized array of centrality scores	Brandes approx. algorithm [8, 43]
Connected Components (CC)	Connectivity	-	$\|V\|$-sized array of component labels	Shiloach-Vishkin [2, 58]

Table 2: Graph inputs and their key properties.

Datasets	Domain	$\|V\|$	$\|E\|$	$\|E\|/\|V\|$
Orkut	social	3,072,626	234,370,166	76
LiveJournal	social	4,847,570	85,702,474	18
CitPatents	citation	6,009,554	33,037,894	6
Twitter	social	61,578,414	2,405,026,390	39
Friendster	social	124,836,179	3,612,134,270	29
Protein	biology	8,745,543	1,309,240,502	149

or rebalancing operations to start. Note that DGAP stores all the locks in DRAM instead of PMs to increase performance. If a crash occurs, all the locks are lost. The pending rebalancing operation will be recovered by checking the *per-thread undo log*. The pending edge writes will be ignored, as they have not yet been returned successfully to users.

4 EVALUATION

We developed DGAP using the PMDK library [50]. Its core data structure consists of approximately 2,000 lines of C++ code. The code is publicly available on Github[1]. In this section, we compare DGAP with other graph analysis frameworks on real-world graphs with synthetic graph insertion patterns. The results reported are the averages of five runs.

4.1 Evaluation Setup

Evaluation Platform. We conducted all evaluations on a Dell R740 rack server equipped with a 2nd generation Intel Xeon Scalable Processor (Gold 6254 @ 3.10 GHz) featuring 18 physical cores. The server also included 6 DRAM DIMMs with 32 GB each (for a total of 192 GB) and 6 Optane DC DIMMs with 128 GB each (for a total of 768 GB). We configured Optane DC in *App Direct* mode. The system ran Ubuntu 20.04 and used the Linux kernel version 4.15.0. Our implementation is based on PMDK 1.12.

Graph Algorithms. To ensure a fair comparison among various graph analysis frameworks, we used the same implementations of four graph algorithms from the GAP Benchmark Suite (GAPBS) [3]. These algorithms are PageRank (PR), Breadth-First Search (BFS), Betweenness Centrality (BC), and Connected Components (CC), detailed in Table 1. GAPBS also offers an optimized Compressed Sparse Row (CSR) implementation, which we modified for persistent memory to serve as one of our evaluation baselines.

Graph Datasets. We used several real-world graphs from SNAP datasets [38] in our evaluations. Table 2 lists these graphs and their key properties. We generate the insertion order by randomly shuffling all the edges for these datasets. Note that, in all the experiments, we will insert the first *10%* of the graph and then start to benchmark the insertion performance for the purpose of warming up the system, similar to the warm-up stage in YCSB [12].

Compared Systems. To showcase the performance of DGAP, we compare it with multiple data structures and state-of-the-art dynamic graph frameworks.

First, we ported two foundational graph data structures to persistent memory to serve as baselines. The **Compressed Sparse Row (CSR)** on persistent memory is based on GAPBS. CSR serves as a baseline for graph analysis evaluations since 1) it can not be updated and 2) it offers the optimal graph analysis performance due to its compact memory layout. We also implemented **Blocked Adjacency-List (BAL)** on persistent memory as another extreme baseline. BAL is known to have poor graph analysis performance due to pointer chasing and great edge insertions performance due to efficient appending to a block. We use BAL as a baseline to understand the insertion performance of DGAP.

We further compared DGAP with three state-of-the-art dynamic graph frameworks designed to support graph updates and analysis. **LLAMA** uses a multi-versioned CSR structure to enable fast graph analysis and graph mutations [42]. The graph updates are conducted in batches and organized as multiple immutable snapshots in LLAMA. To avoid creating too many snapshots, in our evaluation, we only created a snapshot after inserting 1% of the graph, which ranges from 330K edges to 36M edges, depending on the chosen graph dataset. In total, we created 90 snapshots for each graph (the first 10% warm-up is a single snapshot). Because graph analysis in LLAMA can not read the latest graph unless the snapshot is created, these large snapshots mean its graph analysis tasks may miss as many as 36 million edges, which might not be acceptable in some applications. We ported LLAMA to persistent memory by changing the location of its snapshot files to PMs space, which shows a naive way of moving existing graph data structure to persistent memory.

GraphOne is an in-memory graph analysis framework with an extra durability guarantee using external non-volatile devices [33]. New data is first stored in an in-DRAM edge list in an append-only manner. Background threads incrementally move this data to non-volatile memory for persistence. To port GraphOne to persistent memory, we changed the location of *durable phase* to the PM space and required it to flush DRAM data after each 2^{16} insertions to reduce the chances of losing data. We do not limit the DRAM usage of GraphOne during graph analysis. Hence for some graphs, the graph data may be completely cached in DRAM. Due to these settings, we name this baseline as **GraphOne-FD**, indicating GraphOne Flushing-DRAM, in the rest of the paper.

XPGraph is state-of-the-art PM-based dynamic graph system [64]. It is based on GraphOne but extends it with new designs for persistent memory. Specifically, XPGraph stores both the edge list and adjacency list in persistent memory to guarantee data persistence

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

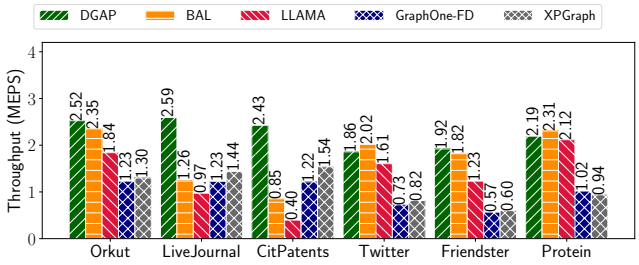


Figure 5: Dynamic graph insertion throughput in million edges per second (MEPS). *Higer value is better.*

and leverages the DRAM as a cache to batch data into the adjacency list. Similar to GraphOne, XPGraph also transfers data to DRAM for graph analysis. In our evaluations, we used the default parameter settings of XPGraph for comparisons.

4.2 Graph Insertions Performance

We first compared the graph updates, particularly the edge insertion performance of DGAP, with other systems. Fig. 5 shows the graph insertion throughput in MEPS (Million Edges Per Second) using a single writer thread. The scalability results are reported later. From these results, we can observe that DGAP achieves almost the best performance across all datasets among all the frameworks. It delivers 1.03 × −2.82× better performance than BAL, which is considered extremely efficient in graph insertions as edges are simply appended to the end of each block. However, the inefficient usage of persistent memory (e.g., journaling and transaction for crash consistency) makes it slower in many cases. DGAP also outperforms LLAMA, GraphOne, and XPGraph on persistent memory by up to 6×, 2.5×, and 2.3×, respectively. It is obvious to us that the costs of asynchronous batch data structure conversions and movements between DRAM and PMs in LLAMA, GraphOne, and XPGraph impact the performance significantly. It is worth noting that, from the results, XPGraph performs better than GraphOne, but not as significant as the original paper reports [64]. This is because our GraphOne-FD has a large batch write size in DRAM, which offers better performance but is impractical as this data may be lost. Still, the better performance of DGAP clearly showcases the efficiency of mutable CSR data structure on persistent memory.

4.2.1 Graph Insertions Scalability. We further evaluated the graph insertions scalability by increasing the number of concurrent writer threads from 1 to 16. Table 3 shows the MEPS throughput of 1, 8, and 16 threads. We can see DGAP scales with more threads. It delivers up to 4.3× throughput in 16 threads compared with single thread case. The concurrency model and write optimizations implemented in DGAP help deliver such a scalable graph insertion performance (6 to 8 million edges/sec), which might be needed in many real-time big data applications.

Across various systems, DGAP consistently ranks as either the best or very close to the best in all scalability cases. BAL occasionally delivers superior performance, primarily due to our implementation

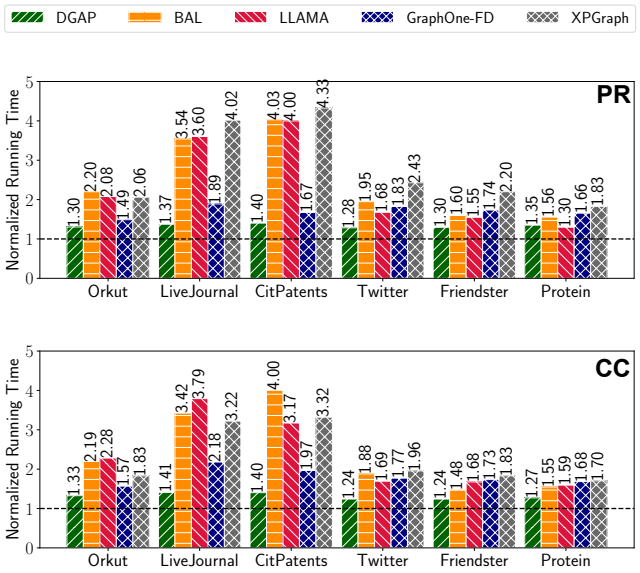


Figure 6: Time to run PageRank (PR) and Connected Components (CC), normalized to CSR on PMM. *Smaller is better.*

of BAL utilizing finer-grain locks for concurrent insertions. Specifically, while DGAP locks writers by edge section, BAL employs vertex-based locking. Consequently, as the number of threads increases, its performance scales more effectively. However, this may not be a realistic representation, as an excessive number of locks are needed. The scalability results of XPGraph are also noteworthy, as it surpasses DGAP in the 16-thread case for three graphs. In fact, these three graphs are all relatively small. We attribute the exceptional performance to XPGraph's design. Specifically, XPGraph includes a circular edge log for temporarily storing new insertions. By default, the circular edge log has a capacity of 8GB, which can entirely accommodate the three smaller graphs: Orkut, LiveJournal, and CitPatents. In this context, archiving is not activated for these graphs, resulting in XPGraph exhibiting exceptional performance. For larger graphs with over a billion edges, DGAP demonstrates 12 − 21% better performance, as XPGraph is compelled to flush the DRAM caches back to the persistent edge list more frequently.

4.3 Graph Analysis Performance

Graph analysis performance is key to our graph frameworks. In this section, we show the performance of running four classic graph algorithms (listed in Table 1) on different graphs. Among these four algorithms, PageRank (PR) and Connected Components (CC) access all vertices in each iteration, while Breadth-First Search (BFS) and Betweenness Centrality (BC) access parts of the graphs each time based on the calculation. They show different access patterns which may impact the performance of the frameworks, as shown below.

1) PageRank (PR) and Connected Components (CC). Fig. 6 illustrates the relative speed of PageRank compared to CSR using a single thread. Compared with CSR, which is best for graph analysis, DGAP introduces only 37% overhead on average and achieves up

Table 3: Graph insertion throughput (MEPS) using the different number of writer threads. *Larger throughput is better.*

	T_1					T_8					T_{16}				
Graph	DGAP	BAL	LLAMA	GO-FD	XPGrp.	DGAP	BAL	LLAMA	GO-FD	XPGrp.	DGAP	BAL	LLAMA	GO-FD	XPGrp.
Orkut	**2.52**	2.35	1.84	1.23	1.30	**6.49**	5.97	2.33	2.54	4.78	7.37	5.26	2.40	2.86	**8.84**
LiveJournal	**2.59**	1.26	0.97	1.23	1.44	**6.27**	4.79	1.07	2.63	5.63	7.95	5.92	1.09	2.94	**11.24**
CitPatents	**2.43**	0.85	0.40	1.22	1.54	**6.82**	3.45	0.41	2.62	6.21	7.23	4.68	0.42	2.81	**12.91**
Twitter	1.86	**2.02**	1.61	0.73	0.82	5.35	**5.51**	2.13	1.99	3.22	**6.82**	5.99	2.17	2.43	6.06
Friendster	**1.92**	1.82	1.23	0.57	0.60	4.29	**5.63**	1.52	2.40	2.51	**6.03**	5.82	1.53	3.35	4.95
Protein	2.19	**2.31**	2.12	1.02	0.94	**7.43**	5.82	3.09	3.21	3.54	**8.30**	6.23	3.18	4.08	6.96

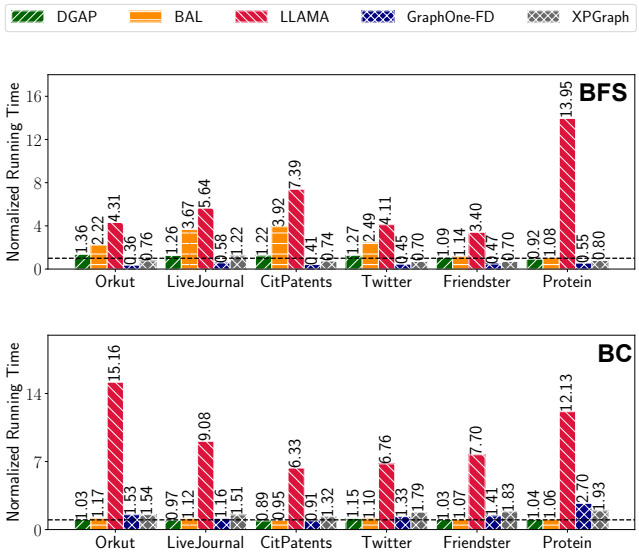


Figure 7: Time to run Breadth-First Search (BFS) and Betweenness Centrality (BC), normalized to CSR on PMM. *Smaller is better.*

to 2.9×, 2.9×, 1.4×, and 3.1× better performance compared to BAL, LLAMA, GraphOne, and XPGraph respectively. It is particularly interesting to observe that DGAP outperforms GraphOne-FD in most datasets, even it is actually running on DRAM-cached data. We believe this is because GraphOne uses adjacency list as its in-memory data structure, which is less efficient for graph analysis tasks that apply to all vertices and edges of the graph. While, since DGAP is a mutable CSR, it shows much better cache locality in running these algorithms such as PageRank. We can observe the same behaviors when running a similar algorithm CC, which iterates all vertices/edges in each iteration. Specifically, Fig. 6 illustrates the relative speed of CC compared to CSR on all the systems. Again, DGAP shows up to 2.9×, 2.7×, 1.6×, and 2.4× better performance than BAL, LLAMA, GraphOne, and XPGraph respectively.

2) Breadth-First Search (BFS) and Betweenness Centrality (BC). Fig. 7 shows the relative speed of Breadth-First Search and Betweenness Centrality compared to CSR. For BFS, DGAP outperforms BAL and LLAMA by 2.30× and 3.71×, respectively on average. However, DGAP performs 2.77× and 1.81× worse than GraphOne and XPGraph in this particular workload. This is expected since BFS is accessing edges of random vertices each time. The adjacency list in GraphOne and XPGraph performs very well for these tasks. CSR can not fully leverage its own spatial locality. In addition, since most BFS only reaches a small part of the graph, GraphOne and XPGraph can successfully cache the graph in DRAM. We observe similar trends for Betweenness Centrality (BC) as Fig. 7 shows. Since BC is more computationally and memory intensive. It also covers larger parts of the graphs during computation, we can see that DGAP actually catches up and delivers similar performance compared with DRAM-based GraphOne and XPGraph. Specifically, DGAP outperforms BAL, LLAMA, GraphOne, and XPGraph by up to 1.08×, 8.19×, 1.21×, and 1.85× respectively.

4.3.1 Graph Analysis Scalability. To examine the scalability of DGAP, we further ran the same graph algorithms using 1 to 16 threads and report the execution time (in seconds) in Table 4. Due to the space limits, we only report results of 1 thread and 16 threads for each case. From these results, we make server observations. First, DGAP scales well. It delivers up to 14.3×, 13.6×, 15.6×, and 4.7× speedup using 16x threads running PageRank, BFS, BC, and CC algorithms respectively. It is interesting to see that DGAP does not scale well in CC. In fact, all the systems do not scale well in this algorithm. After checking the source code, we noticed the bottleneck actually comes from its inappropriate *parallel for* scheduling keywords. If fixed, CC will deliver similar scalability for all frameworks. Since our goal is not to improve the algorithm implementation, we reported the results from the original GAPS implementation. Second, DGAP still delivers the best performance in most graph analysis algorithms. Similar to the single thread case, DGAP performs worse than GraphOne and XPGraph in the BFS case. As discussed earlier, this is mostly because GraphOne and XPGraph run BFS purely in DRAM and their adjacency list structure fits BFS well.

4.4 DGAP Components Evaluations

DGAP Components Evaluations. In DGAP, we introduce three designs to maximize PMs. We further evaluated their contributions to the final performance. Specifically, we implemented and compared three different versions of DGAP by incrementally excluding its key components: (i) removing *per-section Edge Logs* as 'No EL'; (ii) further removing *per-thread Undo Log* as 'No EL&UL', replaced using PMDK transactions; and (iii) further removing Data Placement in DRAM as 'No EL&UL&DP', meaning both vertex array and edge array are on PMs. The graph insertion performance results are reported in Table 5. We only report the results for small-size graphs, as we were not able to finish running all the tests on larger graphs in a reasonable time.

The results show that the *per-section edge log* contributes the most in performance improvements. Without it, DGAP performs

Table 4: The execution time (in seconds) of four algorithms on all systems. T_1 denotes the time of one thread and T_{16} denotes that of 16 threads.

	PageRank												BFS											
	CSR		DGAP		BAL		LLAMA		GraphOne		XPGraph		CSR		DGAP		BAL		LLAMA		GraphOne		XPGraph	
Graph	T_1	T_{16}	T_1	T_{16}	T_1	T_{16}	T_1	T_{16}	T_1	T_{16}	T_1	T_{16}	T_1	T_{16}	T_1	T_{16}	T_1	T_{16}	T_1	T_{16}	T_1	T_{16}	T_1	T_{16}
Orkut	24.18	1.67	31.55	2.21	53.21	3.57	50.24	9.51	36.01	2.63	49.87	3.72	0.33	0.03	0.46	0.04	0.74	0.06	1.44	0.33	0.12	0.01	0.25	0.03
LiveJournal	9.07	0.71	12.46	0.94	32.12	2.30	32.69	5.12	17.14	1.24	36.45	3.04	0.34	0.03	0.43	0.04	1.26	0.10	1.93	0.50	0.20	0.03	0.42	0.05
CitPatents	5.83	0.49	8.17	0.63	23.47	1.73	23.30	2.83	9.75	0.70	25.21	2.38	0.47	0.04	0.57	0.05	1.84	0.14	3.46	0.68	0.19	0.03	0.35	0.06
Twitter	425.11	31.59	545.92	39.30	828.07	56.67	712.73	99.83	775.83	45.10	1032.06	77.99	7.91	0.71	10.09	0.74	19.72	1.47	32.50	6.65	3.58	0.33	5.55	0.71
Friendster	873.38	65.41	1131.84	80.84	1394.05	97.70	1353.57	186.81	1515.38	85.77	1922.26	142.49	14.77	1.12	16.10	1.19	16.79	1.41	50.23	13.54	6.92	0.50	10.41	1.07
Protein	203.48	13.22	274.91	16.85	316.65	20.43	264.23	34.59	336.89	20.61	372.11	27.96	0.90	0.08	0.82	0.08	0.97	0.10	12.51	1.27	0.50	0.04	0.72	0.09

	BC												CC											
	CSR		DGAP		BAL		LLAMA		GraphOne		XPGraph		CSR		DGAP		BAL		LLAMA		GraphOne		XPGraph	
Graph	T_1	T_{16}	T_1	T_{16}	T_1	T_{16}	T_1	T_{16}	T_1	T_{16}	T_1	T_{16}	T_1	T_{16}	T_1	T_{16}	T_1	T_{16}	T_1	T_{16}	T_1	T_{16}	T_1	T_{16}
Orkut	5.22	0.42	5.40	0.42	6.10	0.46	79.07	5.71	7.98	0.58	8.01	0.81	2.60	0.42	3.45	0.73	5.71	0.88	5.94	0.87	4.08	0.75	4.77	0.71
LiveJournal	4.37	0.33	4.23	0.32	4.91	0.36	39.72	2.76	5.06	0.36	6.62	0.61	0.99	0.42	1.40	0.80	3.40	0.87	3.76	1.17	2.16	0.75	3.20	1.03
CitPatents	3.90	0.29	3.49	0.26	3.71	0.27	24.72	1.70	3.54	0.26	5.15	0.47	1.67	0.48	2.34	0.49	6.68	1.43	5.30	2.07	3.28	0.81	5.54	1.68
Twitter	106.10	7.83	122.39	7.86	117.09	8.41	717.39	48.54	141.17	9.13	190.22	15.81	71.53	16.45	88.76	23.48	134.42	28.68	121.06	25.20	126.66	24.80	139.90	30.89
Friendster	203.63	14.70	209.34	14.47	216.92	15.15	1568.58	105.37	287.51	17.59	372.87	28.95	155.40	23.72	192.71	36.41	229.48	33.45	260.48	42.37	269.54	37.79	284.53	44.65
Protein	2.01	0.31	2.09	0.27	2.14	0.27	24.42	1.86	5.43	0.45	3.88	0.47	66.50	4.52	84.52	6.74	102.86	6.74	106.01	11.19	112.01	9.67	113.26	11.93

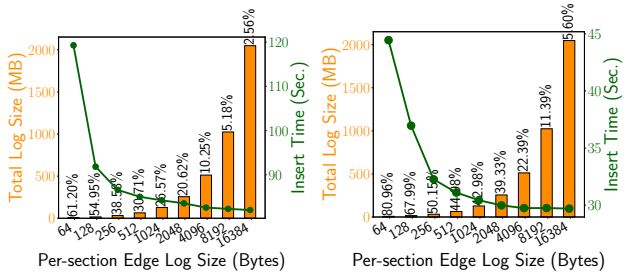
Table 5: Insertion performance (in seconds) of different DGAPs.

Datasets	DGAP	No EL	No EL&UL	No EL&UL&DP
Orkut	83.55	374.86	383.52	588.37
LiveJournal	29.74	136.28	146.09	240.46
CitPatents	12.25	51.26	58.47	107.39

4.5× worse because of the write amplification caused by the nearby shifts. Specifically, with *per-section edge log*, DGAP is able to reduce the write amplification by 6× in the Orkut graph. Additionally, *per-thread undo log* contributes another 13% performance improvement by reducing the high memory allocation and excessive ordering cost of transactions. Finally, placing the *vertex array* in PMs would incur about 2× performance overhead. Placing all the remaining metadata (e.g., PMA tree) in PMs would even double the overhead.

DGAP Configurations Evaluations. Besides three system components, DGAP includes a set of configurations, impacting its performance. For example, the size of *per-section edge log* will affect the PM usages as well as the rebalancing frequency, impacting the insertion performance. To evaluate it, we compared how its size, `ELOG_SZ`, would impact graph insertion performance and PMs consumption. The results are reported in Fig. 8. Due to space limits, we only show results for Orkut and LiveJournal graphs. Other graphs have similar patterns. We changed `ELOG_SZ` from 64 bytes to 16 KB. The bar length represents the total space needed to store all the *per-section edge log*, which increases proportionally as `ELOG_SZ` increase. The labels above each bar further report the percentage-wise utilization of these logs during graph insertions. We can see as the edge log increases, the utilization rate reduces significantly from 80.96% to 5.60% as there might not be so many *nearby shifts* to fill the logs. The green line shows the delivered insertion performance based on each log size. It is clear that larger logs reduce the insertion time. But the benefits become much smaller after 2048, which is chosen as default `ELOG_SZ` size in DGAP.

DGAP Recovery Evaluations. Each time DGAP reboots, it reloads the metadata into DRAM before operating. Such a normal start is fast. In our evaluation, we found that DGAP spends 1.16 seconds in rebooting even on the largest Friendster graph. After crash, DGAP needs to do more housekeeping work to recover system statuses. These steps include scanning the *edge array* and *logs* to recover the inconsistencies caused by the crash. This indicates DGAP crash recovery time will depend on the graph size. However,



(a) Orkut **(b) LiveJournal**

Figure 8: Impacts of the size of per-section edge log.

sequential access in PMs is fast, and so is the DGAP recovery. In our experiment, we found that for the smaller graphs (e.g., Orkut, LiveJournal, and CitPatents), DGAP takes less than 1 second. For the larger graphs, it may take more than 4 seconds. But, note that these time costs are for recovery from a crash only.

5 RELATED WORKS

The works most closely related to ours are NVGRAPH [40] and XPGraph [64]. Both frameworks are designed for persistent memory devices. NVGRAPH proposed a dual-version data structure for NVM and DRAM to achieve high-speed data persistence and graph analysis. However, since NVGRAPH was designed before actual persistent memory devices were released, many of its assumptions have later been shown to be inaccurate [70]. Consequently, it did not leverage many performance features of PMs. As such, we do not compare DGAP with NVGRAPH, as it wouldn't be a fair comparison. Similar to DGAP, XPGraph was designed for and evaluated on Intel Optane PMs, and is essentially a PM-based GraphOne. Through extensive evaluations, we demonstrate that DGAP outperforms XPGraph in both graph updates and graph analysis tasks, highlighting the promising performance of mutable CSR data structures. A recent study [19] systematically benchmarks graph processing on PMs. However, this study assumes that persistent memory functions as volatile, larger DRAM serving only graph analysis, which is fundamentally different from DGAP.

In addition to PM-based graph analysis, there has been a large number of PM-based indexing data structures, such as B+-Tree [9,

10, 23, 39, 41, 47, 71, 74] and Hashtable [6, 11, 35, 45, 76, 77]. Some works [18, 21, 22, 29, 31, 37, 44, 63] also proposed general guidelines for porting in-memory data structures to PMs. Many of the DGAP's design choices are aligned with these existing studies, but focus more on graph updates and analysis.

In addition to PM-based graph frameworks, there are a significant amount of single-node dynamic graph analysis frameworks. We categorize them into in-memory and out-of-core frameworks. For in-memory dynamic graph frameworks [17, 24, 30, 48, 65], their graphs are not persistent and need rebuilding after a crash or reboot. Even with data periodically synchronized to fast non-volatile storage devices, like PMs, existing in-memory graph frameworks still face the challenges in striking a balance between data loss and graph update speed. Our evaluations on BAL, LLAMA, and GraphOne show naively porting existing in-memory graph frameworks to persistent memory will experience performance issues. DGAP roots from in-memory data structure (mutable CSR) as well, but contains a series of new designs to maximize the performance. Existing out-of-core dynamic graph frameworks are designed based on slow block-based storage devices [34, 42]. For example, in LLAMA [42], newly added edges are first batched up in the delta map and periodically synced to a CSR snapshot. Such batch behaviors may not be necessary on persistent memory. While, in DGAP, graph changes are immediately visible to analytic tasks.

6 CONCLUSION AND FUTURE WORK

In this study, we present DGAP, a new graph analysis framework built on persistent memory. DGAP leverages existing DRAM-based mutable Compressed Sparse Row (CSR) graph structure with extensive new designs for persistent memory devices to achieve both efficient graph updates and graph analysis. Our results show DGAP outperforms state-of-the-art dynamic graph frameworks, such as LLAMA, GraphOne, XPGraph on PMs by up to 3.2× in graph updates and 3.77× in graph analysis. Our exploration of DGAP shows that persistent memory is a promising alternative to support efficient dynamic graph analysis. In the future, we plan to further improve DGAP designs, including a Copy-on-Write strategy for Degree Cache and a fine-grained locking mechanism. We also plan to investigate how to extend DGAP to a distributed environment using RDMA in PMs to support even larger graphs.

ACKNOWLEDGMENTS

We sincerely thank the anonymous reviewers for their valuable feedback. This work was supported in part by NSF grants CNS-1852815, CCF-1910727, CCF-1908843, and CNS-2008265.

REFERENCES

[1] Hiroyuki Akinaga and Hisashi Shima. 2010. Resistive random access memory (ReRAM) based on metal oxides. *Proc. IEEE* 98, 12 (2010), 2237–2251.
[2] David A Bader, Guojing Cong, and John Feo. 2005. On the architectural requirements for efficient execution of graph algorithms. In *2005 International Conference on Parallel Processing (ICPP'05)*. IEEE.
[3] Scott Beamer. 2015. GAP Benchmark Suite. https://github.com/sbeamer/gapbs. Accessed July. 30, 2021.
[4] Scott Beamer, Krste Asanovic, and David Patterson. 2012. Direction-optimizing Breadth-First Search. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'12)*.
[5] Michael A Bender and Haodong Hu. 2007. An adaptive packed-memory array. *ACM Transactions on Database Systems (TODS'07)* 32 (2007).
[6] Lawrence Benson, Hendrik Makait, and Tilmann Rabl. 2021. Viper: An Efficient Hybrid PMem-DRAM Key-Value Store. *Proc. VLDB Endow.* 14, 9 (2021).
[7] Maciej Besta, Marc Fischer, Vasiliki Kalavri, Michael Kapralov, and Torsten Hoefler. 2021. Practice of Streaming Processing of Dynamic Graphs: Concepts, Models, and Systems. *IEEE Transactions on Parallel and Distributed Systems* (2021).
[8] Ulrik Brandes. 2001. A faster algorithm for betweenness centrality. *Journal of mathematical sociology* 25, 2 (2001).
[9] Shimin Chen and Qin Jin. 2015. Persistent b+-trees in non-volatile main memory. *Proceedings of the VLDB Endowment* 8, 7 (2015), 786–797.
[10] Youmin Chen, Youyou Lu, Kedong Fang, Qing Wang, and Jiwu Shu. 2020. UTree: A Persistent B+-Tree with Low Tail Latency. *Proc. VLDB Endow.* 13, 12 (2020).
[11] Zhangyu Chen, Yu Hua, Bo Ding, and Pengfei Zuo. 2020. Lock-free Concurrent Level Hashing for Persistent Memory. In *2020 USENIX Annual Technical Conference (USENIX ATC'20)*.
[12] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing (SoCC'10)*. ACM.
[13] Intel Corporation. 2019. Key features on Cascade Lake. https://www.intel.com/content/www/us/en/products/platforms/details/cascade-lake.html. Accessed Jan. 22, 2023.
[14] Intel Corporation. 2021. eADR: New Opportunities for Persistent Memory Applications. https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent-memory-applications.html. Accessed Jan. 22, 2023.
[15] Dean De Leo and Peter Boncz. 2021. Teseo and the Analysis of Structural Dynamic Graphs. *Proc. VLDB Endow.* 14, 6 (2021).
[16] David Ediger, Rob McColl, Jason Riedy, and David A. Bader. 2012. STINGER: High performance data structure for streaming graphs. In *IEEE Conference on High Performance Extreme Computing (HPEC'12)*.
[17] Soukaina Firmli, Vasileios Trigonakis, Jean-Pierre Lozi, Iraklis Psaroudakis, Alexander Weld, Dalila Chiadmi, Sungpack Hong, and Hassan Chafi. 2020. CSR++: A Fast, Scalable, Update-Friendly Graph Data Structure. In *24th International Conference on Principles of Distributed Systems (OPODIS'20)*.
[18] Michal Friedman, Naama Ben-David, Yuanhao Wei, Guy E. Blelloch, and Erez Petrank. 2020. NVTraverse: In NVRAM Data Structures, the Destination is More Important than the Journey. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'20)*.
[19] 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. *Proc. VLDB Endow.* 13, 8 (2020).
[20] Linley Gwennap. 2019. *First Optane DIMMs Disappoint.* The LinleyGroup.
[21] Swapnil Haria, Mark D Hill, and Michael M Swift. 2020. MOD: Minimally ordered durable datastructures for persistent memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*.
[22] Hanxian Huang, Zixuan Wang, Juno Kim, Steven Swanson, and Jishen Zhao. 2021. Ayudante: A Deep Reinforcement Learning Approach to Assist Persistent Memory Programming. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'21)*.
[23] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. 2018. Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree. In *16th USENIX Conference on File and Storage Technologies (USENIX FAST'18)*.
[24] Abdullah Al Raqibul Islam, Dong Dai, and Dazhao Cheng. 2022. VCSR: Mutable CSR Graph Format Using Vertex-Centric Packed Memory Array. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid'22)*.
[25] Abdullah Al Raqibul Islam, Dong Dai, Anirudh Narayanan, and Christopher York. 2020. A Performance Study of Optane Persistent Memory: From Indexing Data Structures' Perspective. In *36th International Conference on Massive Storage Systems and Technology (MSST'20)*.
[26] Abdullah Al Raqibul Islam, Christopher York, and Dong Dai. 2022. A performance study of optane persistent memory: from storage data structures' perspective. *CCF Transactions on High Performance Computing* (24 Sep 2022).
[27] Anand Iyer, Li Erran Li, and Ion Stoica. 2015. CellIQ : Real-Time Cellular Network Analytics at Scale. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI'15)*.
[28] 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).
[29] 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 SIGOPS 28th Symposium on Operating Systems Principles (SOSP'21)*.
[30] James King, Thomas Gilray, Robert M Kirby, and Matthew Might. 2016. Dynamic-CSR: A format for dynamic sparse-matrix updates. In *Springer-Verlag*, Vol. 9697. 61–80.
[31] R. Madhava Krishnan, Wook-Hee Kim, Xinwei Fu, Sumit Kumar Monga, Hee Won Lee, Minsung Jang, Ajit Mathew, and Changwoo Min. 2021. TIPS: Making Volatile

Index Structures Persistent with DRAM-NVMM Tiering. In *2021 USENIX Annual Technical Conference (USENIX ATC'21)*.

[32] Emre Kültürsay, Mahmut Kandemir, Anand Sivasubramaniam, and Onur Mutlu. 2013. Evaluating STT-RAM as an energy-efficient main memory alternative. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'13)*.

[33] Pradeep Kumar and H Howie Huang. 2019. Graphone: A data store for real-time analytics on evolving graphs. In *17th USENIX Conference on File and Storage Technologies (FAST'19)*.

[34] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. Graphchi: Large-scale graph computation on just a PC. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)*.

[35] Kenneth Lamar, Christina Peterson, Damian Dechev, Roger Pearce, Keita Iwabuchi, and Peter Pirkelbauer. 2021. PMap: A Non-volatile Lock-free Hash Map with Open Addressing. In *IEEE 10th Non-Volatile Memory Systems and Applications Symposium (NVMSA'21)*.

[36] Benjamin C Lee, Ping Zhou, Jun Yang, Youtao Zhang, Bo Zhao, Engin Ipek, Onur Mutlu, and Doug Burger. 2010. Phase-change technology and the future of main memory. *IEEE micro* 30, 1 (2010), 143–143.

[37] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. 2019. Recipe: Converting Concurrent DRAM Indexes to Persistent-Memory Indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19)*.

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

[39] Tongliang Li, Haixia Wang, Airan Shao, and Dongsheng Wang. 2022. SSB-Tree: Making Persistent Memory B+- Trees Crash-Consistent and Concurrent by Lazy-Box. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS'22)*.

[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 *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT'19)*.

[41] Jihang Liu, Shimin Chen, and Lujun Wang. 2020. LB+Trees: Optimizing Persistent Index Performance on 3DXPoint Memory. *Proc. VLDB Endow.* 13, 7 (2020).

[42] Peter Macko, Virendra J Marathe, Daniel W Margo, and Margo I Seltzer. 2015. Llama: Efficient graph analytics using large multiversioned arrays. In *IEEE 31st International Conference on Data Engineering (ICDE'15)*.

[43] Kamesh Madduri, David Ediger, Karl Jiang, David A Bader, and Daniel Chavarria-Miranda. 2009. A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. In *IEEE International Symposium on Parallel & Distributed Processing (IPDPS'09)*.

[44] Amirsaman Memaripour, Joseph Izraelevitz, and Steven Swanson. 2020. Pronto: Easy and Fast Persistence for Volatile Data Structures. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*.

[45] Moohyeon Nam, Hokeun Cha, Young ri Choi, Sam H. Noh, and Beomseok Nam. 2019. Write-Optimized Dynamic Hashing for Persistent Memory. In *17th USENIX Conference on File and Storage Technologies (USENIX FAST'19)*.

[46] Optane. 2019. Intel Optane Persistent Memory. https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/optane-dc-persistent-memory-brief.html.

[47] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD'16)*.

[48] Prashant Pandey, Brian Wheatman, Helen Xu, and Aydin Buluc. 2021. Terrace: A Hierarchical Graph Container for Skewed Dynamic Graphs. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD'21)*.

[49] Ali Pinar and Michael T Heath. 1999. Improving performance of sparse matrix-vector multiplication. In *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing (SC'99)*.

[50] pmem.io Persistent. 2019. Persistent Memory Programming. https://pmem.io.

[51] Moinuddin K Qureshi, John Karidis, Michele Franceschini, Vijayalakshmi Srinivasan, Luis Lastras, and Bulent Abali. 2009. Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'09)*.

[52] Moinuddin K Qureshi, Vijayalakshmi Srinivasan, and Jude A Rivers. 2009. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th annual international symposium on Computer architecture (ISCA'09)*.

[53] Simone Raoux, Geoffrey W Burr, Matthew J Breitwisch, Charles T Rettner, Y-C Chen, Robert M Shelby, Martin Salinga, Daniel Krebs, S-H Chen, H-L Lung, et al. 2008. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development* 52, 4.5 (2008), 465–479.

[54] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-stream: Edge-Centric Graph Processing using Streaming Partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP'13)*.

[55] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. 2018. The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing. *Proc. VLDB Endow.* 11, 4 (2018).

[56] Mo Sha, Yuchen Li, Bingsheng He, and Kian-Lee Tan. 2017. Technical report: Accelerating dynamic graph analytics on gpus. *arXiv preprint arXiv:1709.05061* (2017).

[57] Bin Shao, Haixun Wang, and Yatao Li. 2013. Trinity: A Distributed Graph Engine on a Memory Cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD'13)*.

[58] Yossi Shiloach and Uzi Vishkin. 1980. *An O (log n) parallel connectivity algorithm*. Technical Report. Computer Science Department, Technion.

[59] Hongping Shu, Hongyu Chen, Hao Liu, Youyou Lu, Qingda Hu, and Jiwu Shu. 2018. Empirical study of transactional management for persistent memory. In *IEEE 7th Non-Volatile Memory Systems and Applications Symposium (NVMSA'18)*.

[60] Yongju Song, Wook-Hee Kim, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. 2023. Prism: Optimizing Key-Value Store for Modern Heterogeneous Storage Devices. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS'23)*.

[61] Kosuke Suzuki and Steven Swanson. 2015. A survey of trends in non-volatile memory technologies: 2000-2014. In *Proceedings of the IEEE International Memory Workshop (IMW'15)*.

[62] Alexander van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2019. Persistent memory i/o primitives. In *Proceedings of the 15th International Workshop on Data Management on New Hardware (DaMoN'19)*.

[63] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. 2011. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies (USENIX FAST'11)*.

[64] Rui Wang, Shuibing He, Weixu Zong, Yongkun Li, and Yinlong Xu. 2022. XP-Graph: XPline-Friendly Persistent Memory Graph Stores for Large-Scale Evolving Graphs. In *Proceedings of the 55th IEEE/ACM International Symposium on Microarchitecture (MICRO'22)*.

[65] Brian Wheatman and Helen Xu. 2018. Packed Compressed Sparse Row: A Dynamic Graph Representation. In *IEEE High Performance Extreme Computing Conference (HPEC'18)*.

[66] Brian Wheatman and Helen Xu. 2021. A Parallel Packed Memory Array to Store Dynamic Graphs. In *Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX'21)*.

[67] Kai Wu, Jie Ren, Ivy Peng, and Dong Li. 2021. ArchTM: Architecture-Aware, High Performance Transaction for Persistent Memory. In *19th USENIX Conference on File and Storage Technologies (USENIX FAST'21)*.

[68] Yinjun Wu, Kwanghyun Park, Rathijit Sen, Brian Kroth, and Jaeyoung Do. 2020. Lessons Learned from the Early Performance Evaluation of Intel Optane DC Persistent Memory in DBMS. In *Proceedings of the 16th International Workshop on Data Management on New Hardware (DaMoN'20)*.

[69] 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 Seventeenth European Conference on Computer Systems (EuroSys'22)*.

[70] 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 *18th USENIX Conference on File and Storage Technologies (USENIX FAST 20)*.

[71] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. 2015. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. In *13th USENIX Conference on File and Storage Technologies (USENIX FAST'15)*.

[72] J Joshua Yang, Dmitri B Strukov, and Duncan R Stewart. 2013. Memristive devices for computing. *Nature nanotechnology* 8, 1 (2013), 13.

[73] Pantea Zardoshti, Michael Spear, Aida Vosoughi, and Garret Swart. 2020. Understanding and improving persistent transactions on optane™ DC memory. In *34th IEEE International Parallel and Distributed Processing Symposium (IPDPS'20)*.

[74] Bowen Zhang, Shengan Zheng, Zhenlin Qi, and Linpeng Huang. 2022. NBTree: A Lock-Free PM-Friendly Persistent B+-Tree for EADR-Enabled PM Systems. *Proc. VLDB Endow.* 15, 6 (2022).

[75] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. 2018. GraphIt: A High-Performance Graph DSL. *Proc. ACM Program. Lang.* 2, OOPSLA (2018).

[76] Pengfei Zuo, Yu Hua, and Jie Wu. 2018. Write-Optimized and High-Performance Hashing Index Scheme for Persistent Memory. In *13th USENIX Symposium on Operating Systems Design and Implementation (USENIX OSDI'18)*.

[77] Pengfei Zuo, Yu Hua, and Jie Wu. 2019. Level Hashing: A High-Performance and Flexible-Resizing Persistent Hashing Index Structure. *ACM Trans. Storage* 15, 2 (2019).