Enforcing Crash Consistency of Evolving Network Analytics in Non-Volatile Main Memory Systems

***Abstract*—Evolving graph processing has enabled the modeling of many complex network systems, e.g., online social networks and gene networks. Existing in-memory graph data structures cannot effectively exploit the current and ongoing adoption of emerging non-volatile main memory (NVMM) for two reasons. (1) Ephemeral graph data structures are not crash-consistent nor durable for NVMM. Corruption is likely when updating its correlated application-defined data and runtime states in the face of hardware or software failures. (2) NVMM writes and reads may incur higher latency than DRAM. Placing the data structures in NVMM may result in a significant loss of performance.**

In this paper, we propose a novel persistent evolving graph data structure, named NVGRAPH, for both computing and in-memory storage of evolving graphs in NVMM. We devise NVGRAPH as a multi-version data structure, wherein a minimum of one version of its data is stored in NVMM to provide the desired durability at runtime for failure recovery, and another version is stored in both DRAM and NVMM to reduce the NVMM-induced memory latency. We dynamically transform the layout of NVGRAPH including changing the size of its partition in DRAM and the position of its base snapshot exploiting network properties and data access patterns of workloads. For the evaluation of NVGRAPH, we implement four representative real-world graph applications: pagerank, BFS, influence maximization, and rumor source detection. The experimental results show that the performance of NVGRAPH is comparable to other in-memory data structure (e.g., CSR and LLAMA) while using 70% less DRAM. It scales well up to 10 billion edges and 201 snapshots and supports crash consistency. It offers up to the 21X speedup of execution time compared to the scale-up graph computation approaches (e.g., GraphChi and X-stream).

***Index Terms*—Evolving Graphs, Crash Consistency, Non-Volatile Main Memory, Graph Layout Transformation**

I. INTRODUCTION

Evolving graphs consist of multiple snapshots of graphs corresponding to different points in time. Evolving graph processing has enabled modeling of many complex network systems consisting of millions if not billions of nodes, e.g., online social networks [1], gene networks [2], and power flow networks [3], among many others. Applications designed using the evolving graph model have irregular access patterns for traversing a single snapshot or traveling multiple snapshots. Consequently, there is a strong desire to use in-memory graph data structures (e.g., compressed sparse row representation (CSR) [4], LLAMA [5], or Hybrid Graph [6]) to achieve high performance, scalability, as well as high memory efficiency.

The in-memory graph data structures mainly consist of three parts: topology data (i.e., node tables and edge tables), runtime states (e.g., activeness/inactiveness of nodes), and application-defined data (e.g., page ranks of nodes). Memory requirements for storing these data can be much larger than DRAM size. At the same time, it is difficult to scale the DRAM to higher capacities that can match the memory requirements of evolving graph applications because of the associated capital costs and power consumption. Either scale-out or scale-up approaches may be used to overcome this limitation. The existing graph analytics frameworks (e.g., PowerGraph [7], GraphLab [8], and RAMCloud [9]) scale out through partitioning and hosting the graph data in distributed DRAMs of multiple servers in a cluster. The downside of such approaches is that their performance is constrained by networking latency. For scaling up, graph data are stored on secondary storage devices (e.g., solid-state disks [10] or hard disks [11]). Their performance is dramatically limited by I/O capacity of the underlying storage systems.

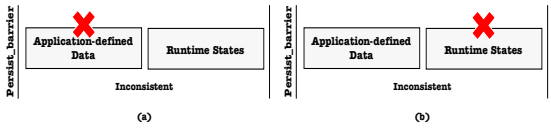


Fig. 1. Inconsistent graph caused by (a) application-defined data write fails and (b) runtime-state write fails.

This paper explores *Non-Volatile Main Memory* (NVMM) (e.g., Memristor [12] and Intel Optane DIMMs [13], [14]) to extend memory capacity for serving large-scale evolving graph applications. However, we found that no existing in-memory graph data structures can take full advantage of NVMM because they do not provide crash consistency in NVMM systems and their memory layout is not aware of the induced-latency of NVMM.

First, **crash consistency** is the recoverability of persistent data from memory in a consistent state after system failures. Graph data structures are not crash-consistent for the following two reasons. (1) *Inconsistent updates of correlated data*: for a large body of graph applications, every write access to a node in NVMM consists of at least two separate write requests to NVMM. For example, in page ranking, each write access to a node requires one write (W_A) to update its rank score and the other one (W_B) to update its activeness for determining the termination of the application. If a system failure occurs after W_A and before the completion of W_B, the memory controller would observe a stale activeness value upon system recovery, introducing data inconsistency in NVMM. Figure 1 demonstrates that a system failure can result in inconsistent application-defined data and runtime states. Similarly, in rumor source detection, each write access to a node requires

	DRAM	NVMM	Flash
Read Latency(ns)	60	**100**	25,000
Write Latency(ns)	60	**150**	200,000
Endurance(writes/bit)	$> 10^{16}$	$10^6 - 10^8$	$10^4 - 10^5$

TABLE I
CHARACTERISTICS OF DRAM AND NVMM. NOTE: THE TABLE CONTENTS ARE BASED MAINLY ON [14], [17]–[19].

one NVMM write (W_C) to update the flag which indicates whether the node was infected and another write (W_D) to store the path of infected nodes. Failures may happen between W_C and W_D. (2) *Partial updates of array elements*: many graph applications use large arrays to store application-defined data and runtime states. If a failure happens before a CPU cache flush writes array elements in NVMM, the array data after failure may not match the normal output without failures. This will cause data inconsistency in NVMM. With the inconsistent data, graph applications cannot directly recover memory states even they are stored in NVMM.

To provide crash consistency, existing graph software saves the checkpoints of application-defined data and runtime states on slow storage devices via I/O bus [5], [15]. The incurred I/O operations can cause severe performance bottleneck and a waste of CPU cycles. Simply storing the data on NVMM-based file systems seems a straightforward approach and requires minimal changes to the applications. However, it can significantly waste the performance benefits of NVMM: (1) it under-utilizes memory resources as essentially two duplicated copies of the application-defined data and runtime states have to be stored in DRAM and NVMM, separately; and (2) it prevents us from using byte-addressability of NVMM to extend memory capacity because the persistent graph data in NVMM are only accessed for failure recovery, not for a program at its normal execution. Applications can also use transaction models (e.g., redo or undo logging mechanisms) to provide crash consistency. However, they may degrade the applications' performance by up to 102% [16].

Second, the layout of existing in-memory data structures is suboptimal for evolving graph computing using both DRAM and NVMM. In the existing systems, graph data were either directly placed in DRAM [5] or stored in the memory cache of slow storage medium (e.g., hard disks [6]). But none has been optimized to leverage NVMM, which has its unique characteristics (as shown in Table I). Ignoring the differences between NVMM and DRAM will have a detrimental impact on the performance of evolving graph applications. (1) The write latency of NVMM is 2.5X greater than that of DRAM [17]–[19]. At the same time, evolving graph applications can be write-intensive for updating application-defined data and runtime states. For the applications related to our research (e.g., rumor source detection in online social networks), memory writes account for at least 50% of the total number of memory accesses to application-defined data and runtime states. Furthermore, as the graph evolves, we can observe that different sets of nodes in networks become more frequently accessed than others while the network property of nodes (e.g., centrality and betweenness) changes over time for changing graph topology. (2) The evolving graph data structures typically consist of a base snapshot and multiple delta snapshots for reducing its memory footprint [5], [6]. The position of the base snapshot determines the number of pointer chasing (memory read) operations that are required to recover a particular graph snapshot. All the existing approaches were proposed to minimize the size of the data structures. However, none of them considered varied memory access cost incurred by the enormous amount of pointer chasing operations in NVMM.

In this paper, we propose a novel data structure, named NVGRAPH, for both graph computing and in-memory storage of evolving graphs. It stores graphs as time series of continuous snapshots. The structure is designed for applications that receive a steady stream of new graph data periodically. To provide crash consistency, NVGRAPH is designed to use multi-version data structures to store application-defined data and runtime states. It enforces crash consistency using NVMM on memory bus, while the existing graph systems rely on accessing slow storage devices on I/O bus. Specifically, the topology data of NVGRAPH consists of one base snapshot and multiple delta snapshots. The first snapshot is created when the first graph is loaded; then each following load creates the other delta snapshots. It stores at least two versions of the correlated application-defined data and runtime states. Its persistent version is immutable and stored in NVMM and can be used for failure recovery upon failures leveraging its non-volatility. In addition, its ephemeral version is stored in both NVMM and DRAM and used for in-memory computing leveraging its byte-addressability of NVMM. To reduce the memory usage of storing multi-version data, NVGRAPH supports data sharing between the two data versions. Consequently, it can guarantee that at least one version of the data is consistent while updating the other one without transactional logging and using special PMEM instructions (e.g., clwb and sfence) [20] for enforcing the ordering of memory writes. NVGRAPH dynamically transforms the layout of its ephemeral version, including the nodes to store in DRAM, the size of in-memory data structures, and the position of its base snapshot to improve DRAM efficiency and hide NVMM-induced memory latency. In summary, we made the following contributions.

- We propose a novel evolving graph data structure NVGRAPH which effectively extends memory capacity using NVMM for large-scale graph computing and provides crash-consistency exploring multi-version data structures.
- We design algorithms using network property (e.g., centrality) and data access patterns of evolving graphs to dynamically transform the layout of NVGRAPH for reducing NVMM-induced latency.
- NVGRAPH provides an easy-to-program interface with which users are freed from error-prone and tedious tasks of persistent pointer management. It explores the non-volatility of NVMM for providing near-instantaneous failure recovery.
- We implemented a software prototype of graph analytics framework using NVGRAPH and its algorithms. Our

evaluation results with real-world graph applications (e.g., rumor source detection in social networks) validate the correctness of the algorithms and show that NVGRAPH scales up to 10 billion edges and 201 snapshots, provides crash consistency, and achieves similar scalability as the existing in-memory data structures.

II. RELATED WORK

As NVMM promises a significant benefit, a wide range of applications and libraries have been developed to address performance and consistency issues caused by NVMM. Some effort closely related to NVGRAPH is discussed below.

A. Multi-Versioned Graph Computation

DeltaGraph built a distributed hierarchical index structure for efficient retrieval of graph snapshots [21]. The index structure is mainly optimized for disks. It can cause extra latency when being directly used for in-memory graph data analysis. Chronos was designed to exploit temporal locality by placing different versions of data of the same nodes together [22]. It uses data arrays and edge arrays as in-memory representation of graphs. It has two main issues: (1) it can achieve suboptimal performance when the data stored in a multi-versioned node cannot fit in the cache line and (2) the graph structure cannot effectively handle graphs with incremental ingest. Vora et al. proposed to use a structure-of-arrays layout to store evolving graphs to exploit temporal locality [23]. The layout may achieve suboptimal performance because it cannot hide the induced-latency of NVMM. In LLAMA, Macko et al. proposed to use a multi-versioned array as the in-memory representation of evolving graphs [5]. It can effectively handle a large number of incremental ingest using the delta snapshots. Most recently, Ju et al. designed Version Traveler to effectively handle version switching [6]. It caches delta snapshots in memory and stores the base snapshots on disks or SSDs. None of them were proposed for NVMM, whose characteristics are significantly different than DRAM or SSDs using flash as shown in Table I. The differences will suffice a new design of the in-memory representations of evolving graphs and special considerations for incurred write/read latency and crash consistency.

B. Scale-Up Graph Computation using Persistent Storage Devices

For general graph computation, many popular systems have been proposed for scaling up on stand-alone machines. For example, GraphChi [11] was one of the first analytic system designed for hard disks. X-Stream was also designed for hard disks, aiming at eliminating random data access from disks [15]. As flash disks become popular, TurboGraph was designed to exploit the parallelism of flash disks [24]. PrefEdge implemented a prefetching scheme to provide high-throughput I/O accesses to flash [25]. FlashGraph is a semi-external memory graph engine with its vertex stored in memory and edge lists stored on SSDs [10]. It aims at achieving performance comparable to in-memory graph engines. Mosaic was designed to explore flash storage and massively parallel coprocessors (e.g., Xeon Phi) for large-scale graph computation [26]. Malicevis et al. studied the placement of graph data in a hybrid memory system with NVMM and DRAM and showed that performance degradation is of great concern when NVMM is directly used in the memory system in a brutal-force manner [27]. GridGraph uses fine-grained level partitioning algorithms to reduce the I/O amount required for computation. NVGRAPH is a new graph engine specifically designed for NVMM on memory bus. Our design goal is to achieve a performance comparable to in-memory graph engines, but also maintain crash consistency, which has never been studied before. No I/O is required for providing a guarantee of crash consistency with NVGRAPH.

C. Durable and Consistent In-Memory Data Structures

Ordinary in-memory graph data structures are *ephemeral* because the old version can be destroyed when a change is made to the graph, leaving only the new one. When an application fails, it is not possible to access the previous versions for recovery using an ephemeral graph. In general, applications designed with ephemeral data structure may use three techniques to ensure data reliability: journaling [28] and leveraging persistent data structure [29] for transactional-oriented applications and checkpointing for non-transactional-oriented applications [30].

Journaling (write-ahead logging) is mostly used by database systems for updating B/B+-Tree or graph data structures. The updates are written to log files sequentially before writing at its primary location. The logs are used to access an old version of a record when transactions fail. The main disadvantage of this technique is *high I/O overhead* because it requires two disk writes for every update. Leveraging multi-version data structures/shadow paging [31], [32], applications use copy-on-write to perform all updates so that the original version of the data is not mutated until the newer version of data becomes persistent. However, its overhead can be high [32], [33], overshadowing its benefit, especially when data structures mainly reside on slow persistent storage devices, e.g., hard disks.

As NVMM is emerging with many attractive features, novel persistent data structures have been proposed to explore its byte-addressability and non-volatility. For example, CDDS B-Tree [17] was designed to provide both durability and consistency and used for implementation of persistent key-value stores. CDDS B-Tree was derived from a multi-version B-Tree data structure [31]. Other researchers have been rethinking the design of B/B+-Tree to embrace NVMM technology when it resides in NVMM. As an example, persistent B+-Tree [18], [34] and NV-tree [35] were proposed to reduce pointer mutations, which are prone to data corruption.

Non-transactional applications: Non-transactional applications include scientific applications and many graph data analytics frameworks (e.g., LLAMA [5] and Chronos [22]). To ensure data durability, they need to periodically save updates to main data structures (e.g., graphs [5], octrees [36], and

arrays [37]) to on-disk representations of the data structures. These files can then be used for failure recovery. Most of these on-disk representations are nearly identical to their corresponding in-memory representations. For large-scale scientific applications, parallel I/Os of writing snapshots can easily overload a disk-based storage system and cause a severe performance bottleneck [30], [38], [39]. Simply replacing disks with NVMM cannot explore its byte-addressability [40]. Caulfield et al. [41] studied the impact of non-volatile memory on scientific applications. However, they did not address any NVMM-specific issues, e.g., data consistency. PM-octree and DPM-octree were implemented to leverage NVMM for large-scale adaptive mesh simulations [42], [43]. In this paper, NVGRAPH is designed to serve two purposes: (1) storing persistent versions of graph data structures in NVMM and (2) extending memory capacity for non-transactional evolving graph computation. It is inspired by the previous work on persistent octree and B-Tree. However, our focus on hiding NVMM-induced read and write latency exploring network properties will require significant changes to the design and impact our implementation of NVGRAPH.

III. DESIGN OF NVGRAPH

The main problem with providing crash consistency for evolving graphs is that each write access to a node consists of at least two separate writes to NVMM: one write to application-defined data A and one write to runtime state R of graphs. But this relationship is not exposed to crash consistency mechanisms. There is no guarantee that the two separate writes to [A, R] will reach the NVMM simultaneously. We propose to use multi-version data structures to replace ephemeral data structures to store A and R. The new data structure is persistent because at least one version of [A, R] will be stored in NVMM and immutable until a newer version becomes persistent. The runtime system will handle how to persistent [A, R] in both DRAM and NVMM. The new data structure is called NVGRAPH. We will discuss its design and basic operations in the following sections.

A. In-Memory Representation of NVGRAPH

Different from most of the previous work focusing on the layout of topology data [5], [6], [22], NVGRAPH manages the layout of topology data, application-defined data, and runtime states. Figure 2 demonstrates the overview of NVGRAPH data structure in DRAM and NVMM. The **topology data** of NVGRAPH G consists of multiple *snapshots*. They are read-only. Each snapshot consists of an index table and an edge table, which are stored as large arrays in NVMM. Thus, the existing topology data are not subject to crash-consistency issues during the execution of graph applications. NVGRAPH manages a *snapshot cache* in DRAM to hide read-latency of accessing the topology data in NVMM.

The correlated **application-defined data** A and **runtime states** R are formed as a tuple $[A, R]$. Their in-memory representations are multi-version large arrays. They will have at least two versions: $[A^{n-1}, R^{n-1}]$ and $[A^n, R^n]$. $[A^{n-1}, R^{n-1}]$

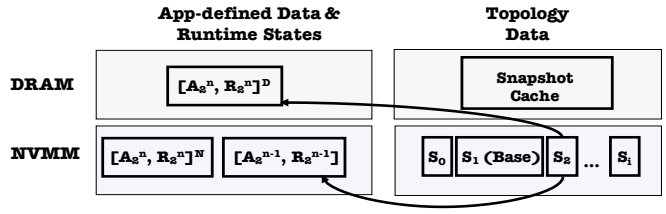


Fig. 2. Overview of the NVGRAPH data structure. NVGRAPH G is partitioned. Its topology data is stored in NVMM and consists of one base snapshot (e.g., S_1) and multiple delta snapshots (e.g., S_0 and S_2). The snapshot cache is used to store the most-frequently accessed snapshots. The correlated application-defined data (e.g., page ranks) and runtime states (e.g., activeness of nodes) for snapshot 2, denoted as $[A_2^{n-1}, R_2^{n-1}]$ and $[A_2^n, R_2^n]$, are generated at step $n-1$ and n respectively. $[A_2^n, R_2^n]$ is further partitioned to $[A_2^n, R_2^n]^D$ stored in DRAM and $[A_2^n, R_2^n]^N$ stored in NVMM.

is a persistent version of the data saved at the end of the computational step[1] $n-1$. $[A^n, R^n]$ is an ephemeral version of the graph being actively accessed at step n. $[A^n, R^n]$ and $[A^{n-1}, R^{n-1}]$ share the corresponding data that are not mutated at step n. No logging or special instructions for enforcing ordering of memory accesses to A and R is needed because our algorithms will guarantee $[A^{n-1}, R^{n-1}]$ is consistent while updating $[A^n, R^n]$. We will develop the basic operations of NVGRAPH in Section III-B.

A graph snapshot S_i in NVGRAPH G consists of topology data T_i, application-defined data A_i, and runtime state R_i. T_i includes a node index table and an edge table in NVMM as demonstrated in Figure 3. One of the snapshots is specified as the base snapshot S_{Base} according to snapshot access frequency. All the nodes and edges in S_{Base} are stored in a variant of compressed sparse row (CSR) representation in NVMM. To reduce memory footprint, other snapshots store only the difference between S_i and S_{i-1} if $Base < i$ or between S_i and S_{i+1} if $Base > i$. Each entry in the index table has two fields: *Snapshot ID* indicating the edge table that stores the adjacency list given a node and *Offset/P* indicating the index into the edge table of the given snapshot or a pointer to the added/deleted nodes in the snapshot. For example, in Figure 3, the first entry $(0, 0)$ in S_1 indicates its first node 0 is shared with S_{Base} and the offset of its adjacency list fragments is 0 in the edge table of S_{Base}. The third entry $(1, P)$ in S_1 indicates that it is a pointer to a new edge 2→0 in the edge table of S_1.

Each entry in the edge table also has two fields: *Node ID* and *Flag* indicating whether the property and runtime states of this edge starting from the node is in DRAM (D) or in NVMM (N). The edge table of S_{Base} is a fixed length array that contains adjacency list fragments stored consecutively. The edge table of non-base snapshot S_i contains only added or deleted nodes in the given snapshot. Each added node in the edge table of S_i ends with a continuation pointer CO : that points to its adjacency list fragment in its neighboring snapshot S_j.

Application-defined data A_i and runtime states R_i for

[1]We assume graph applications consist of a sequence of execution steps, e.g., iterations in calculating the page rank of each node.

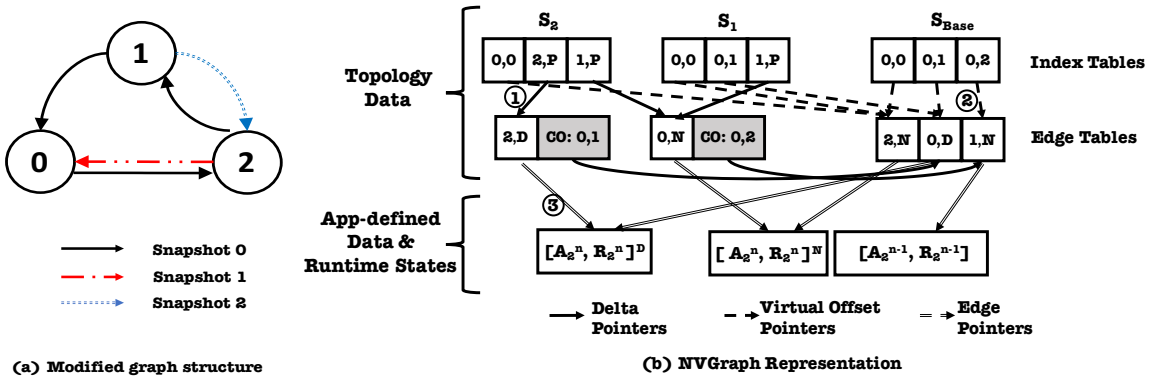


Fig. 3. NVGRAPH representations. (a) An illustration of three snapshots of a graph. A circle represents a node (node id inside) and an arrow represents an edge (edge id omitted). Snapshot 0 consists of solid-arrow edges. Snapshot 1 has one more edge (illustrated by a dashed red arrow). Snapshot 2 has one more edge than Snapshot 1 (illustrated by a dashed blue arrow). (b) An illustration of the evolving graph using NVGRAPH representation. We use pagerank as an example. Its topology data consists of index tables and edge tables. In the edge table, D and N are flags which denote DRAM and NVMM respectively. $(CO:0,1)$ denotes that node 1 has a new edge $1 \rightarrow 2$ in S_2 and the remaining edges of node 1 (e.g., $1 \rightarrow 0$) are shared with S_{Base}. $[A_2^n, R_2^n]^C$ is shared with $[A_2^{n-1}, R_2^{n-1}]$ and not shown in the figure.

the snapshot S_i are represented as a multi-version table in memory. A simple two-version NVGRAPH is described in the paper. $[A_i^{n-1}, R_i^{n-1}]$ were actively accessed at step $n-1$ and became persistent at the end of the step. At step n, every update to $[A_i^{n-1}, R_i^{n-1}]$ results in mutating $[A_i^n, R_i^n]$. No special instructions for enforcing crash consistency is needed because our algorithm can guarantee that $[A_i^{n-1}, R_i^{n-1}]$ is consistent while updating $[A_i^n, R_i^n]$. While $[A_i^{n-1}, R_i^{n-1}]$ is entirely stored in NVMM, $[A_i^n, R_i^n]$ includes three components: $[A_i^n, R_i^n]^D$, $[A_i^n, R_i^n]^N$, and $[A_i^n, R_i^n]^C$, where D, N, C denotes DRAM, NVMM, and common nodes respectively. $[A_i^n, R_i^n]^D$ consists of frequently accessed data and is stored in DRAM. $[A_i^n, R_i^n]^N$ consists of less frequently accessed data and is stored in NVMM. $[A_i^n, R_i^n]^C$ is also stored in NVMM and consists of read-only data that can be shared with $[A_i^{n-1}, R_i^{n-1}]$. For applications that have a significant amount of overlapping data between $[A_i^{n-1}, R_i^{n-1}]$ and $[A_i^n, R_i^n]$, such as when a majority of nodes become inactive, this compact layout of multi-version A and R will reduce their memory usage by up to 83% while providing the guarantee of crash consistency for graph applications used in the paper.

NVGRAPH manages a snapshot cache in DRAM to hide NVMM read latency. The data stored in the cache is delta snapshots. If the cache is full, it must destage the least valuable delta snapshot in the cache to make room for the new snapshot. We will use LRU (Least Recently Used) replacement policy to select the least valuable snapshots.

Figure 3(a) shows a simple 3-version graph. Figure 3(b) shows its NVGRAPH representation in memory. To further reduce the memory footprint of storing the index tables of multiple snapshots, NVGRAPH merges them into a single index table and shares it among the three snapshots, while each snapshot has its own edge table. There are three types of pointers in NVGRAPH. *Delta pointers* point to the added/deleted nodes in the edge tables of delta snapshots. *Virtual offset pointers* are array index that points to the first node in the adjacency list of a given node. And *edge pointers* connect an edge to its associated application-defined data and runtime states. For example, in Figure 3, NVGRAPH uses the delta pointer ① to connect node 1 to its added edge $1 \rightarrow 2$ in S_2. It uses the virtual offset pointer ② to indicate that node 1 is the first node that is adjacent to node 2 in the edge table of S_{Base}. Finally, it uses the edge pointer ③ to connect the edge $1 \rightarrow 2$ to its edge properties stored in $[A_2^n, R_2^n]$ of snapshot 2. The application-defined data and runtime states are stored in either NVMM or DRAM. For example, for S_{Base}, the application-defined data corresponding to edge $0 \rightarrow 2$ (denoted as $(2, N)$ in the edge table) is stored in NVMM, while the data corresponding to edge $1 \rightarrow 0$ (denoted as $(0, D)$) is stored in DRAM because node 1 might be more frequently-accessed than node 0. The layout optimization of $[A^n, R^n]$ is described in Section III-C.

B. Major Operations of NVGRAPH

Insertion and deletion of nodes: NVGRAPH adds nodes to a snapshot by creating a new index table and an edge table. Because most of the nodes will be shared between the new snapshot and the base snapshot, we only need to change the index table where node mutations happen. To delete a node, we mark the node as "deleted" in the index table. It does not require deleting any data because deletion can cause a long write latency just as an insertion for NVMM. The real deletion is handled by garbage collection.

Graph loading and flattening: NVGRAPH can load graph data in edge-list formats. It sorts the input edges before loading and builds index tables and edge tables. When the previous snapshots are no longer needed, users can use the flattening operations of NVGRAPH to create a single and merged snapshot and use it as the base snapshot for further data ingest.

Garbage collection (GC): As the size of NVGRAPH grows and deleted nodes are not freed in NVMM, GC is required to be executed periodically to ensure memory efficiency. Furthermore, we track the percentage of available NVMM space, when it is smaller than a threshold, GC will also be executed on demand.

Data merging and persistence: When step n is completed and a persistent point of the graph needs to be saved. NVGRAPH will merge $[A_i^n, R_i^n]^D$ and $[A_i^n, R_i^n]^C$ with $[A_i^n, R_i^n]^N$ and then mark all the data in $[A_i^{n-1}, R_i^{n-1}]$ as "deleted". The GC routines executed later will delete the data. In the end, we need to swap $[A_i^{n-1}, R_i^{n-1}]$ and $[A_i^n, R_i^n]^N$. Only after this point, the new persistent $[A_i^{n-1}, R_i^{n-1}]$ becomes available and the next step begins. If the applications fail during merging, the data marked as "deleted" will be recovered and used to restart the program. To ensure data consistency, GC is disabled during data merging.

Wear leveling: NVMM has much higher write endurance than flash disks. However, if there are hot data being frequently accessed, it may still wear out after a large number of writes to a single cell. We will use both application-level and system-level approaches to address this issue. At the application level, we periodically re-layout the NVGRAPH data structure so that hot data which are frequently read/written are stored in DRAM. We will discuss it in Section III-C. In addition, we plan to leverage existing system approaches [33], [44], [45] to protect NVMM.

Changing the base snapshot: NVGRAPH uses the most-frequently access (hot) snapshot as the base snapshot to reduce the pointer-chasing operations in NVMM. For this purpose, it tracks the access frequency of the snapshots. If the access frequency of a snapshot S_i is significantly and consistently higher than the current base snapshot, it will choose S_i as the new base snapshot and trigger the operation of changing the base snapshot by merging the snapshot from S_0 to S_i and producing the new delta data for all the non-base snapshots. NVGRAPH will execute the base changing operations when the system is in a light load and in the background, thus minimizing the interference to its front-end workloads. In the experiments, NVGRAPH monitors the system load and executes system maintenance operations (e.g., base changing and GC) when CPU usage is lower than 30%.

Data recovery: After rebooting the crashed compute nodes, NVGRAPH checks the sanity of the pointers to $[A_i^{n-1}, R_i^{n-1}]$ and $[A_i^n, R_i^n]^N$ at the pre-defined NVMM addresses. Then it marks the data in $[A_i^n, R_i^n]^N$ as "deleted" for recycling. Finally, it uses $[A_i^{n-1}, R_i^{n-1}]$ to restore $[A_i^n, R_i^n]^D$, $[A_i^n, R_i^n]^N$, and $[A_i^n, R_i^n]^C$ incrementally and on demand.

C. Data Partitioning using Network Properties

NVMM has longer read/write latency than DRAM. We dynamically change the layout of NVGRAPH to hide the latency. Specifically, we study how to partition the application-defined data and runtime states $[A, R]$ according to network properties and how to allocate DRAM between $[A_i, R_i]^D$ and snapshot cache for memory efficiency in this section.

For snapshot S_{Base}, NVGRAPH partitions its $[A_{Base}, R_{Base}]$ into $[A_{Base}, R_{Base}]^D$ in DRAM to store frequently accessed data and $[A_{Base}, R_{Base}]^N$ in NVMM to store less frequently accessed data given the constraint that the size of $[A, R]^D$ should be smaller than the DRAM usage threshold $Size_{DRAM}$. For non-base snapshots, we only incrementally update the partition considering incoming ingest. We use the following steps to select the hot data to store in DRAM.

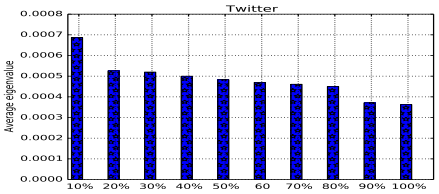


Fig. 4. The relationship of the access frequency of nodes and its eigenvector centrality value. The nodes are sorted based on their memory access frequency and separated into 10 percentage groups.

Step 1: Node selection. We select a subset of nodes that are more influential than others. To this end, we need to design metrics to measure the importance of a node. We use a simple heuristic based on network centrality [46] because it is strongly correlated to the memory access frequency of a node in graph computation. For example, in the Twitter network [47], we profiled the memory access frequency of nodes and computed their eigenvalue centrality when executing rumor-source detection algorithm [1]. We grouped the nodes according to their memory access frequency. We found that the top 10% of mostly-visited nodes also have a consistently larger value of eigenvalue centrality than the nodes in other percentile groups (e.g., 20% and 30%), as shown in Figure 4. Consequently, we can use the eigenvalue centrality to identify frequently accessed nodes in graphs.

Step 2: Edge selection. We randomly choose a few influential edges from the adjacency list of the selected nodes that have a significantly large value of eigenvalue centrality. The weighted random sampling approach is used to produce higher accuracy. Furthermore, we observe that a large number of common neighbors indicate that the edge could be frequently accessed by the nodes in a community. Therefore, we use the reverse of Jaccard co-efficient of neighborhoods as a metric to quantify the importance of an edge $i \rightarrow j$. NVGRAPH only selects the edges whose value of the reverse of Jaccard co-efficient is significantly larger than an empirical threshold.

Step 3: Incremental updates. We run the node and edge selection algorithms only when a new base snapshot is created. We save the results (e.g., centrality values of nodes and Jaccard co-efficient of edges) on persistent storage devices. With incoming ingest, we update the values considering the added/deleted nodes and edges incrementally to reduce the overhead of computing the values of network properties [48].

D. Determining the Size of Snapshot Cache

Because NVGRAPH stores both $[A_i^n, R_i^n]^D$ and snapshot cache in DRAM, we need to determine the sizes of the two data structures so that topology data in snapshot cache and the application-defined data and runtime states in $[A_i^n, R_i^n]^D$ can all be accessed efficiently. In this paper, we build a memory access model considering the cache hit ratio and memory access latency and use the model to determine the size of snapshot cache and $[A_i^n, R_i^n]^D$ at runtime.

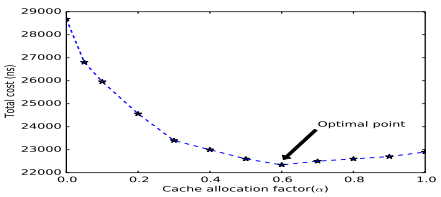


Fig. 5. The relationship between the total cost and the memory allocation factor α.

$$Cost_{total} = Cost_{AR} + Cost_T \tag{1}$$

Let's assume the DRAM size is S_{DRAM}, the snapshot cache size is S_{cache}, and the size of $[A_i^n, R_i^n]^D$ is S_{AR}. We also use α to denote the DRAM allocation factor. Then $S_{cache} = \alpha * S_{\text{DRAM}}$ and $S_{AR} = (1-\alpha) * S_{\text{DRAM}}$. The total cost $Cost_{total}$ of memory access can be modeled using Equation 1, where $Cost_{AR}$ is the cost of accessing A and R and $Cost_T$ is the cost of accessing the topology data. We then assume the snapshot cache hit ratio is $Hit_{cache}(\alpha)$ and the hit ratio in $[A_i^n, R_i^n]^D$ for reading $[A, R]$ is $Hit_{AR}(\beta)$ ($\beta = 1-\alpha$) given a DRAM allocation factor α.

$$Cost_T = \frac{1}{p-q+1} * \sum_{i=p}^{q} Num_{S_i}^{Read} * [Hit_{cache}(\alpha) * Lat_D^{Read} + (1 - Hit_{cache}(\alpha)) * (Lat_N^{Read} + Lat_D^{Write})] \tag{2}$$

$$Cost_{AR} = Num_{AR}^{Write} * Lat_D^{Write} + Num_{AR}^{Read} * [Hit_{AR}(\beta) * Lat_D^{Read} + (1 - Hit_{AR}(\beta)) * Lat_N^{Read}] \tag{3}$$

We then model $Cost_T$ using Equation 2 where $Num_{S_i}^{Read}$ is the total number of reads for a snapshot. An application can access multiple of them (i.e., from snapshot p to q). Lat_D^{Read} is the average DRAM read latency, Lat_N^{Read} is the average NVMM read latency, and Lat_D^{Write} is the average DRAM write latency. The value of these latency parameters can be found in Table I. We model $Cost_{AR}$ using Equation 3 where Num_{AR}^{Write} and Num_{AR}^{Read} are the total number of writes to $[A_i^n, R_i^n]$, respectively. As a result, when we increase α, $Hit_{cache}(\alpha)$ is increased. Thus, $Cost_{cache}$ is decreased. However, $Hit_{AR}(\beta) = Hit_{AR}(1-\alpha)$ will be decreased accordingly, therefore, increasing $Cost_{AR}$. Our goal is to find α that result in a minimum $Cost_T$.

We instrument the data structure and the cache to provide $Num_{S_i}^{Read}$, Num_{AR}^{Write}, and Num_{AR}^{Read} at runtime. To estimate the cost, we also need to know $Hit_{cache}(\alpha)$ and $Hit_{AR}(1-\alpha)$. We build cache hit ratio curve offline using the four representative graph applications discussed in Section IV. As an example, assuming that we have an access cost graph as shown in Figure 5 we can determine that when α is equal to 0.6, NVGRAPH obtains the minimum cost of accessing the heterogeneous memory devices. As a result, it will allocate 60% of DRAM to the snapshot cache and the rest to $[A_i^n, R_i^n]^D$.

IV. EVALUATION

We implemented NVGRAPH in C++ as a library. Users need to create an NVGRAPH object using nvgraph_create() and call nvgraph_persistent() to create a persistent version of graph in NVMM at the end of an execution step. Users have the control of the granularity/frequency of creating a persistent snapshot. NVGRAPH supports a general-purpose programming model as LLAMA. Users can iterate over all nodes or a set of nodes in the node table. Given a node, users can access its edges in the adjacency list. We implemented four graph applications using a vertex-centric model for evaluation. We plan to add edge-centric implementation of the applications in our future work.

We conduct an extensive performance study for NVGRAPH to experimentally answer the following questions.

- What is the performance of NVGRAPH compared to existing in-memory and out-of-core graph systems?
- Is NVGRAPH scalable for real-world evolving graph applications and is it effective to reduce failure-recovery time?
- What are the performance implication of NVGRAPH when the parameters (e.g., the size of $[A_i^n, R_i^n]^D$ and number of snapshots) change? And how quickly can NVGRAPH move across snapshots and change base snapshots?
- What is the impact of dynamic transformation of NVGRAPH on the reduction of NVMM write latency?

A. Experimental Setup

Computer server for NVMM emulation: we evaluated NVGRAPH on a computer server, which is configured with Intel Xeon E5-2650 3.16GHz CPU (24 cores) and 256 GB DRAM. The server runs Ubuntu Linux operating system 18.04 with kernel-4.15.0. Each node is configured with one hard disk (Western Digital Blue 1 TB) for hosting OS and one SSD (Samsung SSD 850 PRO 256 GB) for hosting the input datasets. We model NVMM using DRAM on the server using an emulation based approach. Our emulator is similar to those used in other projects [49]–[52]. Specifically, our NVMM emulator introduces extra latency for NVMM write and read in routines that write to or read from DRAM. The delay is determined using the worse-case read/write latency in published data in [14], [17]–[19]. We summarized the parameters in Table I. We create delays using a software spin loop [50], [52] that uses the x86 RDTSP instruction to read the processor timestamp counter and spins until the counter reaches the intended delay. We use up to 150 GB DRAM to emulate NVMM.

Graph datasets: We use three graph datasets in the experiments. (1) The *LiveJournal* dataset has 4.8 million nodes and 69 million edges [53]. In the graph, each journal page is a node and an edge $u \rightarrow v$ represents a hyperlink from page u to page v. (2) The *Twitter* dataset has 41.7 million nodes and 1.37 billion edges [47], [54]. In the graph, a user is a node. An edge $u \rightarrow v$ means that user u mentioned v in a tweet. (3)

The *R-MAT* dataset is a synthetic graph which uses a recursive matrix model to generate realistic graph datasets [55]. We use it to evaluate the scalability of NVGRAPH with different graph size (i.e., number of nodes and degrees of nodes).

Each evolving graph used in the following experiments has 21 snapshots unless otherwise specified. We used the three graph datasets to generate its corresponding evolving graphs. Given a graph dataset, we first randomly loaded 80% of its edges to build the base snapshot. Then we randomly selected 1% of its remaining edges and assigned them to each delta snapshot until the remaining 20 snapshots were initialized.

Applications: We implemented four graph applications for benchmarking NVGRAPH. (1) *PageRank:* The application outputs a probability distribution which represents the relative importance of web pages in networks [56]. It executes a random walk which jumps to a random node with a certain probability α, and follows a randomly chosen outgoing edge of a node with probability $1 - \alpha$ from the current node. (2) *Breadth-first search (BFS):* The benchmark traverses a graph from an arbitrary node of the graph. It explores all of the neighbor nodes at the present level before moving on to nodes at the next level. (3) *Influence maximization (Influence):* The application selects a set of most influential users in social networks [57]. It uses the independent cascade model for information propagation. In the model, when a node u first becomes active, it is considered contagious. It has one chance of influencing each inactive neighbor v with probability $f_{u,v}$, independently of the history. If the tentative succeeds, v becomes active. We manually assign $f_{u,v}$ as the weight of edge $u \rightarrow v$ in the networks. (4) *Rumor source detection (Rumor):* The diffusion of malicious rumors makes us vulnerable to various risks. The application is designed to find a set of rumor candidates using k-Minimum Distance Rumor Source Detection (k-MDRSD) algorithm [1] in online social networks. The output of the application can then be used for final investigation of rumor sources.

Graph systems used in the experiments: The four graph applications are implemented using 5 different graph systems, respectively, for a comprehensive evaluation. Specifically, LLAMA [5], NVGRAPH, and Green-Marl [58] are designed for in-memory computing. GraphChi [11] and X-Stream [15] are designed for out-of-core computing using disks or SSDs. In the experiments, we replace disks and SSDs with NVMM to study their performance in a computing environment configured with NVMM. GreenMarl and X-Stream were designed for computation using static graphs. They do not support multi-snapshots of graphs.

Systems and its storage options: (1) *LLAMA (DRAM)*: it uses large multiversioned array (LAMA) and edge tables to store data in DRAM. (2) *NVGRAPH*: it is the multi-version persistent data structure which stores data in both DRAM and NVMM as described in the paper. (3) *LLAMA (NVMM)*: It uses the LAMA data structure, which is the same as LLAMA (DRAM). We modified the LLAMA library to use emulated NVMM instead of DRAM. (4) *GraphChi*: it was designed to store graph data in on-disk $shards$, which store all the edges that have destination in disjoint intervals. We used the data structure as the original implementation described in the GraphChi paper. We use NVMM-based $tmpfs$ file system instead of disks as the storage backend of GraphChi. As a result, the graph data is stored in NVMM and read to DRAM for computing using file-system interface. (5) *Green-Marl*: it uses CSR as the in-memory data structure to store graph data in NVMM. (6) *X-Stream*: it uses stream buffers as the in-memory data structure to store pre-processed edge data in DRAM. Its graph data is stored in NVMM-based tmpfs.

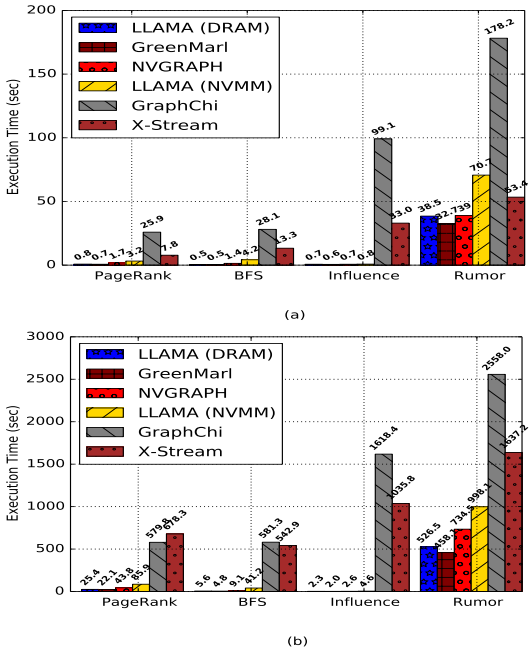
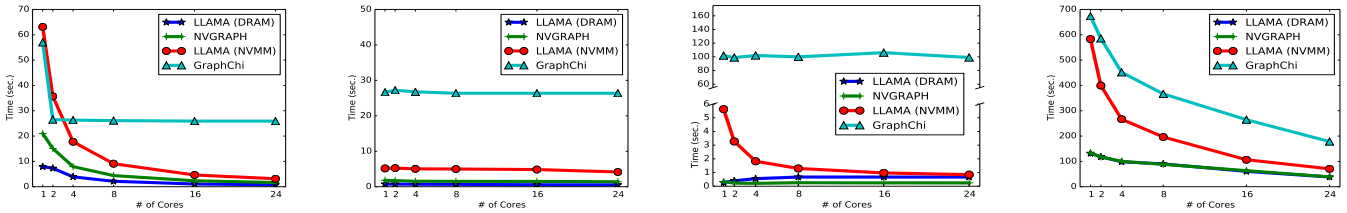


Fig. 6. Performance of NVGRAPH compared to LLAMA, GreenMarl, GraphChi, and X-Stream for (a) LiveJournal and (b) Twitter datasets.

B. Comparison to Other Systems

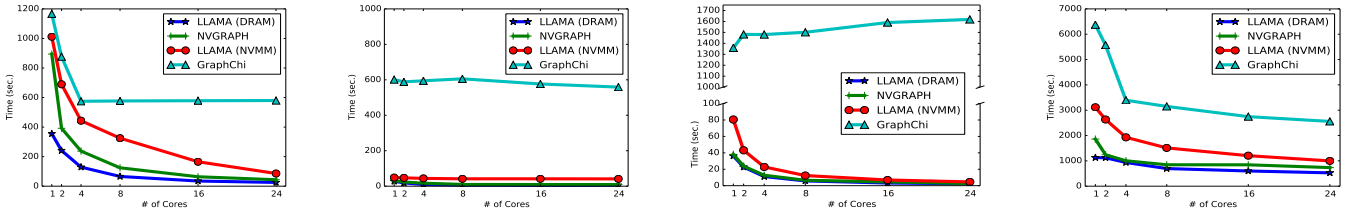
We compare the performance of NVGRAPH to other graph systems. In the experiments, we used 24 cores if the systems support multi-threading using OpenMP or pthreads library. The applications accessed a single snapshot. When NVGRAPH is used, the size of $[A_i^n, R_i^n]^D$ is set to 180 MB and 3.6 GB for LiveJournal and Twitter graphs respectively. The rest is used for the snapshot cache. LLAMA (DRAM) and GreenMarl store all the graph data in DRAM. Their maximum memory usage is 14.7 GB and 15.4 GB, respectively. LLAMA (NVMM) stores the 14.7 GB data in NVMM. For the Twitter graph, the out-of-core systems GraphChi and X-Stream used up to 13.8 GB and 18.2 GB DRAM and 23.0 GB and 17.6 GB NVMM respectively. The results are summarized in Figure 6.

Compared to in-memory graph systems, we have three observations. (1) The execution time of NVGRAPH is 47% smaller than that of LLAMA (NVMM) on average because it stores $[A_i^n, R_i^n]$ and snapshot cache in DRAM to hide NVMM-induced latency. For example, for the LiveJournal datasets, it reduces the number of writes by 30%, 30%, 17%, and 55% for PageRank, BFS, Influence Maximization, and



(a) PageRank (b) BFS (c) Influence Maximization (d) Rumor Source Detection

Fig. 7. The execution time of benchmarks as we increased the number of cores from 1 to 24 for the LiveJournal graph.



(a) PageRank (b) BFS (c) Influence Maximization (d) Rumor Source Detection

Fig. 8. The execution time of benchmarks as we increased the number of cores from 1 to 24 for the Twitter graph.

Rumor Source Detection, respectively. (2) Its performance is 71.3% slower than that of LLAMA (DRAM), which stores the whole graph data in DRAM. However, for *Influence* and *Rumor*, nodes with larger centrality values tend to have higher memory access frequency. When NVGRAPH uses this network property to explore the locality of memory accesses, its execution time is reduced by up to 43% and 45% for Influence Maximization and Rumor Source Detection respectively and the results are very close to those systems using DRAM only. (3) GreenMarl outperforms LLAMA (DRAM) and NVGRAPH. It also uses CSR as the in-memory data structure. Because LLAMA (DRAM) and NVGRAPH use index tables and continuation pointers for the management of multiple snapshots, their memory access latency is longer for pointer chasing to access delta snapshots and application-defined data in NVMM.

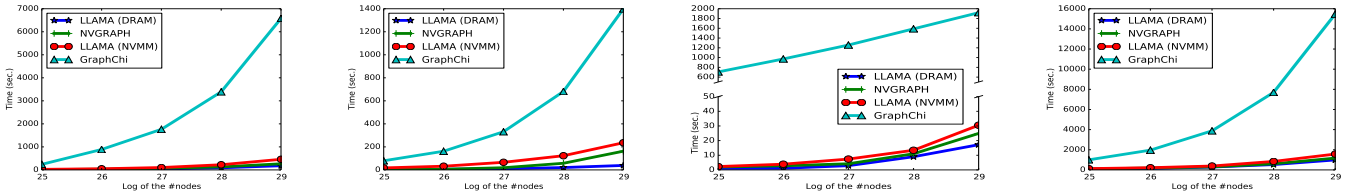
Compared to out-of-core graph systems, we have the following observations. NVGRAPH outperforms both GraphChi and X-Stream even though the latter two systems accessed graph data in an NVMM-based file system. For example, it can reduce the execution time of Influence Maximization and Rumor Source Detection by up to 78% and 37% respectively. The reason is that NVGRAPH uses a variant of CSR as the in-memory data structure, which is more compact and byte-addressable. As a comparison, the basic I/O unit is a *shard* in GraphChi. It needs to repeatedly read a single shard multiple times for random walking operations, which are used extensively in Influence Maximization and Rumor Source Detection. As an example with the Twitter dataset, GraphChi read 34.5 GB and wrote 11.5 GB for Influence Maximization. It read 36.8 GB and wrote 13.8 GB for Rumor Source Detection. Consequently, I/O operations of reading shards slowed down the applications. Another reason is that the I/Os through tmpfs file systems were slow due to higher kernel overhead than direct memory access used in in-memory graph systems, e.g., NVGRAPH and LLAMA.

C. *Scalability of* NVGRAPH

In this section, we first study the scalability of NVGRAPH as we increase the number of cores from 1 to 24. The applications accessed a single snapshot. Figure 7 and Figure 8 show the execution time for the LiveJournal and Twitter graph. Because the performance trend of GreenMarl is similar to LLAMA (DRAM) and X-Stream is similar to GraphChi, we did not show their scalability here. We can observe that NVGRAPH scales well compared to LLAMA (DRAM) and LLAMA (NVMM) for both of the graphs. For NVGRAPH, when the number of cores is increased from 1 to 24, its execution speedup of PageRank, BFS, Influence Maximization, and Rumor Source Detection is on average 21X, 3.6X, 15X, and 2.5X respectively. We observe that the cache miss ratio of the applications using NVGRAPH and LLAMA are 45.4% and 45.5%, respectively, while the cache miss ratio with the out-of-core approaches is 61.7% on average.

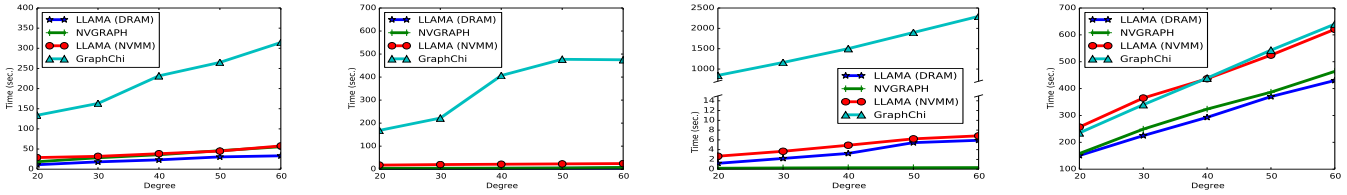
The performance of BFS and Rumor Source Detection is not sensitive to the number of cores because (1) the time ratio of atomic operations (e.g., ATOMIC_ADD) with regard to the total execution time of the applications is increased from 8% with one core to 35% with 24 cores, making it less scalable; (2) the size of parallelizable code is small. GraphChi scales well with a small number of cores for PageRank and Rumor Source Detection. Its curve levels off when the number of cores is larger than 8. The reason is that GraphChi uses the Linux buffer cache to alleviate I/O overhead. However, the buffer cache is shown to have limited scalability with multi-core CPUs [59]. Another observation is NVGRAPH outperforms LLAMA(DRAM) with the LiveJournal graph, i.e., for Influence Maximization. It is because of the overhead of maintaining data pages in LLAMA.

In the second experiment, we study the scalability of NVGRAPH as we increase the number of nodes and the average node degree of the graph respectively. Figure 9 shows the execution time of applications as we increase the number



(a) PageRank (b) BFS (c) Influence Maximization (d) Rumor Source Detection

Fig. 9. The execution time of benchmarks as we increase the number of nodes for the R-MAT graph.



(a) PageRank (b) BFS (c) Influence Maximization (d) Rumor Source Detection

Fig. 10. The execution time of benchmarks as we increase the number of node degree for the R-MAT graph.

of nodes in the graph from 2^{25} to 2^{29} and fix the average degree of the nodes to 16. Figure 10 shows the execution time of applications as we increase the number of degree of nodes from 20 to 60 and fix the total number of nodes to 2^{25}. We used R-MAT to generate the synthetic graphs using probabilities $a = 0.57$, $b = 0.19$, $c = 0.19$. These parameters are recommended by Graph500 [60] to produce graphs with the scale-free property found in real-world graphs. In this experiment, the largest graph has 100 million nodes and 10 billion edges. The results show that NVGRAPH scales as well as other in-memory graph systems as we increase the number of nodes and node degrees. In contrast, the execution time of applications using GraphChi is increased super-linearly, showing its poor scalability even using NVMM-based file systems as storage backends. Another observation is that the performance of NVGRAPH is very close to that of LLAMA (DRAM) for Influence Maximization and Rumor Source Detection. This is because the layout transformation using centrality reduces the execution time of the applications by 36% on average.

D. Effectiveness of Dynamic Transformation

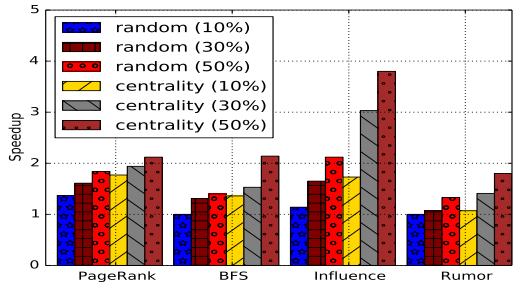


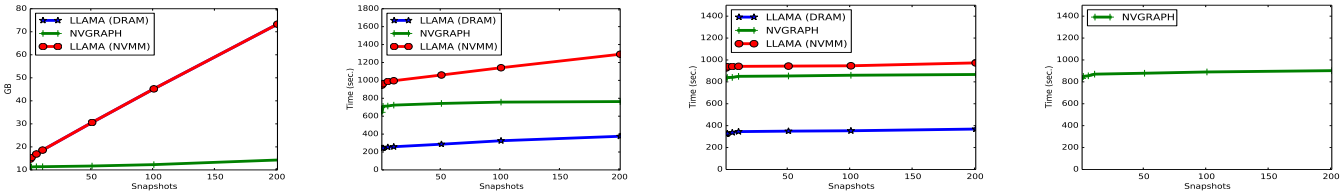
Fig. 11. Execution time speedup of graph applications using dynamic transformation with randomly selected nodes and selection using network centrality. The number in the parentheses indicates the ratio of nodes stored in DRAM.

NVGRAPH uses dynamic transformation to move frequently accessed nodes from NVMM to DRAM, therefore, reducing NVMM-induced memory latency. In this section, we evaluate the effectiveness of dynamic transformation of NVGRAPH with two approaches. In the first approach, we randomly select the nodes from input graphs and store them in DRAM ($random$). In the second approach, we select the nodes according to their eigenvalue centrality ($centrality$). In the experiments, we increased the ratio of nodes in DRAM from 10% to 30% to show the impact of DRAM size. The application accessed a single snapshot. Figure 11 shows the execution time speedup compared to the system that stores 100% of nodes in NVMM.

We have three observations. (1) Dynamic transformation according to node's centrality outperforms that using random selection. For example, for Influence Maximization, its execution time using centrality is reduced by up to 44% because the total number of writes and reads in NVMM is reduced by 2.2X and 30% respectively. (2) The execution time of graph applications is reduced by 25% on average as we increased the DRAM size of $[A_i^n, R_i^n]^D$ from 10% to 50%. This is because it reduces the number of merging operations and the number of NVMM accesses with more nodes being stored in DRAM. (3) The improvement ratio of the speedup varies across applications. As an example, the maximum speedup of Influence Maximization is 3.8X while that of PageRank is only 2.1X on average. After profiling, we found that Influence Maximization has higher temporal locality than PageRank. Its average access frequency of nodes in DRAM is 36 compared to 13 for PageRank.

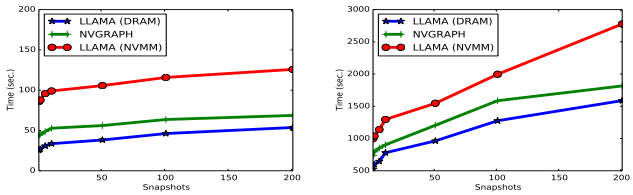
E. Performance of Multi-Snapshot Access

NVGRAPH stores evolving graphs as one base snapshots and multiple delta snapshots. We evaluate its multi-snapshot support in this section. In the experiments, we used the Twitter dataset to generate N snapshots where N= 1, 2, 6, 11, 51, 101, and 201. We first randomly selected 80% of the edges to create



(a) Total Memory Usage (b) Loading (c) Flatten (d) Rebasing

Fig. 12. The time of loading, flattening, and rebasing multiple snapshots of the Twitter graph. Because LLAMA does not support rebasing, we did not present its results in (d).



(a) PageRank (b) Rumor Source Detection

Fig. 13. Execution time across multiple snapshots for the Twitter graph. We did not present the results of BFS and Influence Maximization as they show similar trends.

the base snapshot and then assigned the remaining 20% of the edges to the remaining $N-1$ snapshots. Using this approach, we simulate the small ingests after an initial large ingest to a store of evolving graphs. Because GraphChi has consistently worse performance than both LLAMA and NVGRAPH as shown in Section IV-C, we exclude its measurements in the following results.

We show the execution time of major operations of NVGRAPH including loading, flattening, changing base snapshot (rebasing), as well as the total memory (DRAM + NVMM) usage in Figure 12. NVGRAPH has a smaller memory usage than LLAMA because the latter uses two-level node indexing table based on data pages to reduce I/O cost. We are able to greatly simplify the design of index tables in NVGRAPH to take advantage of the non-volatility and byte-addressability of NVMM, thus reducing its memory usage for maintaining the data pages. The loading, flattening, and rebasing time of NVGRAPH is 61.8% longer than LLAMA (DRAM) for the induced-latency of NVMM.

We show the execution time of the graph applications across multiple snapshots in Figure 13. The execution time of NVGRAPH was increased by 0.38 sec/snapshot when the number of snapshots was increased from 1 to 11. Its time was increased by 0.04 sec/snapshot when the number of snapshots was increased from 11 to 201. This is because the size of delta snapshots became much smaller and the time of accessing the delta data is reduced accordingly.

F. Failure Recovery

System Setting	LLAMA (DRAM)	NVGRAPH (1 core)	NVGRAPH (24 cores)	GraphChi
Time (sec)	356.3	43.8	2.1	1105.6

TABLE II
TIME SPENT ON THE RECOVERY OF GRAPH STATES.

In this section, we compare the time to restart the graph applications after failures using LLAMA, GraphChi, and NVGRAPH. In the experiments, we run PageRank with the Twitter dataset. We sent a SIGTERM signal at t=10 sec to the application and forced it to execute failure recovery handlers in the programs. Specifically, for LLAMA (DRAM), the handler loads the Twitter dataset to produce the evolving graph in memory. For GraphChi, it pre-processes the dataset to generate shards. And for NVGRAPH, it marks the data in $[A_i^{n-1}, R_i^{n-1}]$ as "deleted". Then it uses $[A_i^{n-1}, R_i^{n-1}]$ to restore $[A_i^n, R_i^n]^C$ and $[A_i^n, R_i^n]^N$ on demand. Table II shows the execution time spent on the recovery of graph states. Compared to LLAMA (DRAM), the recovery time with NVGRAPH is reduced from 356.3 sec to 2.1 sec (99%). The speedup of using NVGRAPH is 8X and 25X compared to LLAMA (DRAM) and GraphChi respectively because NVGRAPH operates at memory bandwidth speed while the other two need to read the graph data stored in NVMM based file system. We also observe that NVGRAPH (24 cores) outperforms NVGRAPH (1 core) because with the support of multi-threading NVGRAPH can further speed up the operations (e.g., marking "deletion") in failure recovery with excellent scalability using threading.

V. CONCLUSION

In this paper, we describe the design and implementation of NVGRAPH, a novel crash-consistent evolving graph data structure, for both computing and in-memory storage of evolving graphs in NVMM. NVGRAPH uses a multi-version graph data structure, wherein a minimum of one version of its data is stored in NVMM to provide the desired durability at runtime for failure recovery, and another version is stored in both DRAM and NVMM to reduce the NVMM-induced memory latency. To make the system truly effective, the layout of NVGRAPH is dynamically transformed according to network properties and data access patterns of workloads. A software prototype of NVGRAPH is developed and used to implement four representative real-world graph applications. Our experimental results show that the performance of NVGRAPH is comparable to other in-memory data structures (e.g., CSR and LLAMA) but with 70% less DRAM usage in its execution. Compared to the out-of-core graph system (e.g., GraphChi, X-stream), it offers up to the 21X speedup of execution time. Furthermore, it scales well up to 10 billion edges, 201 snapshots, and 24 cores on a single server. Finally, it provides crash consistency and reduces the failure recovery time of graph applications by up to 99% when NVGRAPH is used and multi-threading support is enabled.

REFERENCES

[1] S. Lim, J. Hao, Z. Lu, X. Zhang, and Z. Zhang, "Approximating the k-minimum distance rumor source detection in online social networks," in *2018 27th International Conference on Computer Communication and Networks (ICCCN)*, 2018.

[2] X. Zhang, U. Khanal, X. Zhao, and S. Ficklin, "Making sense of performance in in-memory computing frameworks for scientific data analysis: A case study of the spark system," *JPDC*, vol. 120, 2018.

[3] G. Liu, X. Chen, Z. Wang, R. Dai, J. Wu, C. Yuan, and J. Tan, "Evolving graph based power system ems real time analysis framework," in *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2018, pp. 1–5.

[4] R. Pearce, M. Gokhale, and N. M. Amato, "Multithreaded asynchronous graph traversal for in-memory and semi-external memory," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11. [Online]. Available: https://doi.org/10.1109/SC.2010.34

[5] P. Macko, V. J. Marathe, D. W. Margo, and M. I. Seltzer, "Llama: Efficient graph analytics using large multiversioned arrays," in *2015 IEEE 31st International Conference on Data Engineering*, ser. ICDE'15, 2015.

[6] X. Ju, D. Williams, H. Jamjoom, and K. G. Shin, "Version traveler: Fast and memory-efficient version switching in graph processing systems," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, 2016.

[7] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, 2012.

[8] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab: A framework for machine learning and data mining in the cloud," *Proc. VLDB Endow.*, vol. 5, no. 8, pp. 716–727, Apr. 2012. [Online]. Available: https://doi.org/10.14778/2212351.2212354

[9] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang, "The ramcloud storage system," *ACM Trans. Comput. Syst.*, vol. 33, no. 3, pp. 7:1–7:55, Aug. 2015. [Online]. Available: http://doi.acm.org/10.1145/2806887

[10] D. Zheng, D. Mhembere, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay, "Flashgraph: Processing billion-node graphs on an array of commodity ssds," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, ser. FAST'15, 2015.

[11] A. Kyrola, G. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a PC," in *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. Hollywood, CA: USENIX, 2012, pp. 31–46. [Online]. Available: https://www.usenix.org/conference/osdi12/technical-sessions/presentation/kyrola

[12] J. J. Yang and R. S. Williams, "Memristive devices in computing system: Promises and challenges," *J. Emerg. Technol. Comput. Syst.*, vol. 9, no. 2, May 2013.

[13] D. X. Memory, https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html.

[14] "Intel optane dimms," https://blocksandfiles.com/2018/12/13/intel-confirms-optane-dimm-and-ssd-speed/.

[15] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: Edge-centric graph processing using streaming partitions," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: ACM, 2013, pp. 472–488. [Online]. Available: http://doi.acm.org/10.1145/2517349.2522740

[16] H. Elnawawy, M. Alshboul, J. Tuck, and Y. Solihin, "Efficient checkpointing of loop-based codes for non-volatile main memory," in *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2017, pp. 318–329.

[17] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell, "Consistent and durable data structures for non-volatile byte-addressable memory," in *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, ser. FAST'11, 2011.

[18] S. N. Shimin Chen, Phillip B. Gibbons, "Rethinking database algorithms for phase change memory," in *CIDR'11: 5th Biennial Conference on Innovative Data Systems Research*, 2011.

[19] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ser. ISCA '09, 2009.

[20] "Persistent memory programming," https://pmem.io/.

[21] U. Khurana and A. Deshpande, "Efficient snapshot retrieval over historical graph data," in *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, ser. ICDE'13, 2013.

[22] W. Han, Y. Miao, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, W. Chen, and E. Chen, "Chronos: A graph engine for temporal graph analysis," in *Proceedings of the Ninth European Conference on Computer Systems*, ser. EuroSys'14, 2014.

[23] K. Vora, R. Gupta, and G. Xu, "Synergistic analysis of evolving graphs," *ACM Trans. Archit. Code Optim.*, vol. 13, no. 4, pp. 32:1–32:27, Oct. 2016. [Online]. Available: http://doi.acm.org/10.1145/2992784

[24] S. Ko and W.-S. Han, "Turbograph++: A scalable and fast graph analytics system," in *Proceedings of the 2018 International Conference on Management of Data*, ser. SIGMOD '18. New York, NY, USA: ACM, 2018, pp. 395–410. [Online]. Available: http://doi.acm.org/10.1145/3183713.3196915

[25] K. Nilakant, V. Dalibard, A. Roy, and E. Yoneki, "Prefedge: Ssd prefetcher for large-scale graph traversal," in *Proceedings of International Conference on Systems and Storage*, ser. SYSTOR'14, 2014.

[26] S. Maass, C. Min, S. Kashyap, W. Kang, M. Kumar, and T. Kim, "Mosaic: Processing a trillion-edge graph on a single machine," in *Proceedings of the Twelfth European Conference on Computer Systems*, ser. EuroSys '17. New York, NY, USA: ACM, 2017, pp. 527–543. [Online]. Available: http://doi.acm.org/10.1145/3064176.3064191

[27] J. Malicevic, S. Dulloor, N. Sundaram, N. Satish, J. Jackson, and W. Zwaenepoel, "Exploiting nvm in large-scale graph analytics," in *Proceedings of the 3rd Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, ser. INFLOW'15, 2015.

[28] R. Hagmann, "Reimplementing the cedar file system using logging and group commit," in *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, ser. SOSP '87, 1987.

[29] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan, "Making data structures persistent," *J. Comput. Syst. Sci.*, vol. 38, no. 1, Feb. 1989.

[30] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate, "Plfs: A checkpoint filesystem for parallel applications," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09, 2009.

[31] P. J. Varman and R. M. Verma, "An efficient multiversion access structure," *IEEE Transactions on Knowledge and Data Engineering*, vol. 9, no. 3, pp. 391–409, May 1997.

[32] C. Mohan, "Repeating history beyond aries," in *Proceedings of the 25th International Conference on Very Large Data Bases*, ser. VLDB '99, 1999.

[33] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better i/o through byte-addressable, persistent memory," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, ser. SOSP '09, 2009, pp. 133–146.

[34] S. Chen and Q. Jin, "Persistent b+-trees in non-volatile main memory," *Proc. VLDB Endow.*, vol. 8, no. 7, Feb. 2015.

[35] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He, "Nv-tree: Reducing consistency cost for nvm-based single level systems," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, ser. FAST'15, 2015.

[36] "Gerris: a tree-based adaptive solver for the incompressible euler equations in complex geometries," *Journal of Computational Physics*, vol. 190, no. 2, pp. 572 – 600, 2003.

[37] W. X. Wang, Z. Lin, W. M. Tang, W. W. Lee, S. Ethier, J. L. V. Lewandowski, G. Rewoldt, T. S. Hahm, and J. Manickam, "Gyro-Kinetic simulation of global turbulent transport properties in tokamak experiments," vol. 13, no. 9, 2006, p. 092505.

[38] X. Zhang, K. Davis, and S. Jiang, "IOrchestrator: Improving the performance of multi-node I/O systems via inter-server coordination," in *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'10)*, 2010.

[39] ——, "Qos support for end users of i/o-intensive applications using shared storage systems," in *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2011, pp. 1–12.

[40] S. Kannan, A. Gavrilovska, K. Schwan, and D. Milojicic, "Optimizing checkpoints using nvm as virtual memory," in *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, 2013, pp. 29–40.

[41] A. M. Caulfield, J. Coburn, T. Mollov, A. De, A. Akel, J. He, A. Jagatheesan, R. K. Gupta, A. Snavely, and S. Swanson, "Understanding the impact of emerging non-volatile memories on high-performance, io-intensive computing," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10, 2010.

[42] B. Nguyen, H. Tan, and X. Zhang, "Large-scale adaptive mesh simulations through non-volatile byte-addressable memory," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC'17, 2017.

[43] B. Nguyen, H. Tan, K. Davis, and X. Zhang, "Persistent octrees for parallel mesh refinement through non-volatile byte-addressable memory," *IEEE Transactions on Parallel & Distributed Systems*, p. 1. [Online]. Available: doi.ieeecomputersociety.org/10.1109/TPDS.2018.2867867

[44] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ser. ISCA '09, 2009.

[45] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A durable and energy efficient main memory using phase change memory technology," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ser. ISCA '09, 2009.

[46] M. E. J. Newman, *Networks : An introduction*, 2010.

[47] "476 million twitter tweets," https://snap.stanford.edu/data/twitter7.html.

[48] B. Bahmani, A. Chowdhury, and A. Goel, "Fast incremental and personalized pagerank," *Proc. VLDB Endow.*, vol. 4, no. 3, pp. 173–184, Dec. 2010. [Online]. Available: http://dx.doi.org/10.14778/1929861.1929864

[49] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," *SIGPLAN Not.*, vol. 47, no. 4, pp. 91–104, Mar. 2011.

[50] J. Ou, J. Shu, and Y. Lu, "A high performance file system for non-volatile main memory," in *Proceedings of the Eleventh European Conference on Computer Systems*, ser. EuroSys '16, 2016.

[51] J. Huang, K. Schwan, and M. K. Qureshi, "Nvram-aware logging in transaction systems," *Proc. VLDB Endow.*, vol. 8, no. 4, Dec. 2014.

[52] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, and M. M. Swift, "Aerie: Flexible file-system interfaces to storage-class memory," in *Proceedings of the Ninth European Conference on Computer Systems*, ser. EuroSys '14, 2014.

[53] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan, "Group formation in large social networks: Membership, growth, and evolution," in *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '06. New York, NY, USA: ACM, 2006, pp. 44–54. [Online]. Available: http://doi.acm.org/10.1145/1150402.1150412

[54] H. Kwak, C. Lee, H. Park, and S. Moon, "What is twitter, a social network or a news media?" in *Proceedings of the 19th International Conference on World Wide Web*, ser. WWW '10. New York, NY, USA: ACM, 2010, pp. 591–600. [Online]. Available: http://doi.acm.org/10.1145/1772690.1772751

[55] D. Chakrabarti, Y. Zhan, and C. Faloutsos, *R-MAT: A Recursive Model for Graph Mining*, pp. 442–446. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/1.9781611972740.43

[56] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Technical Report 1999-66, November 1999, previous number = SIDL-WP-1999-0120. [Online]. Available: http://ilpubs.stanford.edu:8090/422/

[57] F. Bonchi, "Influence propagation in social networks: A data mining perspective," in *2011 IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology*, vol. 1, Aug 2011, pp. 2–2.

[58] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun, "Green-marl: A dsl for easy and efficient graph analysis," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII. New York, NY, USA: ACM, 2012, pp. 349–362. [Online]. Available: http://doi.acm.org/10.1145/2150976.2151013

[59] "A parallel page cache: IOPS and caching for multicore systems," in *Presented as part of the 4th USENIX Workshop on Hot Topics in Storage and File Systems*. Boston, MA: USENIX, 2012. [Online]. Available: https://www.usenix.org/conference/hotstorage12/workshop-program/presentation/Zheng

[60] "Graph 500," https://graph500.org/.