FTGraph: A Flexible Tree-based Graph Store on Persistent Memory for Large-Scale Dynamic Graphs

Gan Sun[a,b,c], Jiang Zhou[a,b,c,*], Bo Li[a,b,c], Xiaoyan Gu[a,b,c], Weiping Wang[a]
[a]*Institute of Information Engineering, Chinese Academy of Sciences*
[b]*School of Cyber Security, University of Chinese Academy of Sciences*
[c]*Key Laboratory of Cyberspace Security Defense*
Beijing, China
{sungan,zhoujiang,libo,guxiaoyan,wangweiping}@iie.ac.cn

Shuibing He
College of Computer Science and Technology, Zhejiang University
Hangzhou, China
{heshuibing}@zju.edu.cn

2024 IEEE International Conference on Cluster Computing (CLUSTER) | 979-8-3503-5871-1/24/$31.00 ©2024 IEEE | DOI: 10.1109/CLUSTER59578.2024.00011

***Abstract*—Traditional in-memory graph systems often suffer from scalability due to the limited capacity and volatility of DRAM. Emerging non-volatile memory (NVM) provides an opportunity to achieve highly scalable and high-performance graph stores for its large capacity and persistence characteristics. However, directly deploying current in-memory graph storage systems on NVM would cause significant inefficiencies in NVM access, as their graph organization designed for DRAM may incur higher write amplification, crash inconsistency and costly concurrency control overhead in NVM for write-intensive workloads. In this paper, we propose FTGraph, a Flexible Tree-based Graph storage system, for both efficient dynamical graph updates and analysis. To achieve this goal, we introduce a novel degree-aware suffix bit tree to effectively manage vertices and edges of the graph, enabling adaptability to real-world power-law degree distributions while significantly reducing NVM writes. Based on it, we adopt two optimization methods, logical vertex ID translation and sequential storage, for vertices with very high degrees within the tree to enhance graph analysis operations. We further integrate 8B NVM atomic writes with optimistic version-based concurrency control through a dual bitmap design to ensure low-overhead, log-free crash consistency and reduce read-writer contention. Experimental results show that FTGraph achieves up to 21.2× higher update performance and up to 85.4× higher analysis performance, compared with state-of-the-art dynamic graph systems implemented on NVMs.**

***Index Terms*—dynamic graphs processing; non-volatile memory; storage system optimization.**

I. INTRODUCTION

In recent years, graph-structured data from the Internet and social media has grown both in size and complexity. For instance, Facebook manages about 1.39 billion active users, and their social network graph has more than 400 billion edges [1]. Many graph processing systems have been proposed recently to enable high-performance graph computation for these dynamically changing graph data [2, 3, 4, 5, 6]. Typically, graph processing involves updating the input edges to graph data structures and running the graph analysis algorithms. That is, graph updates and analysis on dynamic massive graphs should be performed efficiently.

In terms of rapid edge insertions, which are among the most frequent operations in graph updates, it is essential to design a data structure that is scalable, high-performing, and flexible enough to support fast modifications. The recent work GraphTinker [4] employs two widely-adopted hashing schemes (Robin Hood Hashing and Tree-Based Hashing [7]) to form a balanced tree structure that reduces probe distance in following edges, thus supporting high-performance graph updates. While this representation is effective for updates, it lacks the ability to capture patterns of graph analysis. For instance, PageRank and graph traversal analyze a static snapshot of the graph data, with performance depending on the efficiency of scanning vertex neighborhoods. To accelerate graph analysis, most graph processing systems represent the graph using the popular CSR (Compressed Sparse Row) format, as CSR uses sequential memory layouts for adjacent edges, leading to a small storage footprint and high cache efficiency. It achieves rapid scans but is limited in supporting high-throughput updates because each update requires creating a new CSR representation. Therefore, a high-performance graph processing system should balance efficient edge updates and fast algorithm analysis.

With the rapid expansion of graph scale, new challenges arise concerning graph construction and updates, as they spend the most time in existing systems compared to graph analysis performance. For instance, on 32 cores/threads, updating the graph takes up to 90% of the overall running time in incremental connected components [8]. Moreover, efficiently managing hub vertices is an important factor affecting graph update performance, yet most existing graph data structures do not adequately address this aspect. The hub-vertices are a small number of vertices with very high degrees in graphs that follow a power-law distribution [5]. This characteristic has the potential to limit performance and scalability. For instance, GraphTinker creates same-sized EdgeblockArrays as the primary data structure for storing edges associated with a source vertex. This method can lead to extensive pointer chaining within edge blocks of hub vertices, resulting in non-compact edge storage, severe memory wastage, and poor data locality for low-degree vertices. Consequently, a well-designed degree-aware graph store is essential to minimize the number of accesses to DRAM for hub vertices and reduce storage and search overheads for low-degree vertices.

Furthermore, as graphs grow larger, the memory require-

*Corresponding Author

2168-9253/24/$31.00 ©2024 IEEE
DOI 10.1109/CLUSTER59578.2024.00011

ments for storing this data can exceed the size of DRAM, posing scalability challenges for in-memory graph systems due to the constrained capacity of DRAM. To overcome this limitation, many graph storage systems use distributed DRAMs on multiple servers to store graph data. However, frequent network communication is required between hosts for data transfer, and such communication may become a performance bottleneck. Some other graph systems use secondary storage devices (e.g., hard disks or SSDs), but they are hindered by the poor I/O performance of the storage media and do not facilitate efficient random accesses.

Recently, emerging byte-addressable non-volatile memory (NVM) technologies, such as Intel's 3D XPoint Optane persistent memory [9], has brought great performance potential for dynamic graphs. For NVM, its latency is 2-3× higher than DRAM [10], but orders of magnitude lower than hard disks and SSDs. Additionally, a dual-socket machine can be equipped with up to 12 NVDIMMs or up to 6TB NVMs for storage capacity. Such features make it feasible to construct large dynamic graphs entirely on NVMs, which improves performance and scalability while also ensuring data persistence.

However, existing in-memory graph data structures cannot fully leverage NVM if directly ported to it, as NVM and DRAM possess different performance characteristics. First, NVM read-write bandwidth is asymmetric, with the read bandwidth being about 3× higher than that of the write [10]. Existing DRAM-based dynamic graph systems tend not to consider the high amount of writes because DRAM has a smaller gap (1.3×) between read and write bandwidth. Intensive writes in DRAM-based graph systems may introduce dramatic performance degradation on NVM. Second, for NVM, the DRAM-based graph data structure is not crash-consistent and will not recover to a consistent state due to the absence of data consistency enforcement after a system power failure. Thus, it is non-trivial to develop consistent data structures on NVM graph store.

Besides guaranteeing crash consistency, it is also challenging to achieve efficient concurrent access (linearizability) at the same time. Typically, the program creates a critical section to ensure linearizability and then uses persistent instructions (clwb and sfence) to guarantee crash consistency in the critical section. However, it is far from efficient as read-only operations are blocked, and the coarse-grained lock mechanism results in more thread-waiting overhead on NVM compared to DRAM. This is because the read/write latency of NVM is relatively higher than DRAM, and the persistent instructions are costly compared to other instructions.

To address the above challenges, we propose a novel persistent memory graph data structure, named FTGraph, for both efficient graph updates and analysis. In FTGraph, adjacent edges of each vertex are stored separately in the degree-aware suffix bit trees. The degree-aware suffix bit tree is a lightweight and dedicated data structure designed for NVM. It leverages the suffix bits of adjacent vertex IDs to decide the position of edges within the tree, thus enabling fast graph updates while maintaining its own balance effectively. As the tree-based structure might cause expensive random memory jumps during adjacent edge scans and may not fully exploit Intel Optane memory's preference for sequential accesses, we introduce an optimization method for sequential adjacent edge scans. In addition, we integrate 8B (8-byte) NVM atomic writes with optimistic version-based concurrency control through a dual bitmap design in every tree node. Since each bitmap can be updated with a single 8B write instruction, maintaining data consistency after a crash is easily achieved. The dual bitmap is also a natural version lock for concurrent control. The thread only starts the procedure if two bitmaps are the same. The thread reads the new version after the procedure and then restarts it if the two versions differ. Therefore, a dual bitmap design can extract more concurrency, guaranteeing that any concurrent execution remains similar to sequential execution, even in the event of a system crash.

In summary, we make the following contributions:

- We propose a high-performance persistent memory graph store named FTGraph for large-scale dynamic graph updates and analysis. FTGraph adopts a novel degree-aware suffix bit tree to store adjacent edges for each vertex, significantly enhancing graph update performance through an adaptive structure that leverages the power-law distribution of the graph. We also introduce two optimization methods for graph analysis, including logical vertex ID translation and sequential storage, for high levels of the degree-aware suffix bit tree.
- We integrate 8B NVM atomic writes with optimistic version-based concurrency control through a dual bitmap design to reduce the extra overhead of maintaining crash consistency and the read-writer contention.
- We implement the prototype FTGraph on NVMs and conduct extensive experiments to demonstrate its update and analysis efficiency. Results show that FTGraph outperforms the state-of-the-art dynamic graphs by up to 21.2× in graph updates and 85.4× in graph analysis.

II. BACKGROUND AND MOTIVATION

A. Non-volatile Memory

While research on NVM began more than 10 years ago, it wasn't until 2019 that Intel released the first commercially available Optane persistent memory, an innovative new tier between DRAM and SSD. NVM combines byte addressability and persistence to bring greater capacity to the memory bus, overcoming the capacity limitations of traditional DRAM.

Performance characteristics. The performance characteristics of NVM differ from DRAM. The read/write latency of NVM is up to 2–5× higher than DRAM due to its relatively moderate read/write bandwidth [10]. NVM also exhibits asymmetric read/write performance and higher random access overhead. For example, the Intel Optane persistent memory can deliver up to 40 GB/s for sequential reads and 10 GB/s for sequential writes, while dropping to 7.4 GB/s and 5.3 GB/s for random reads and writes, respectively [11]. Therefore, it is important to minimize unnecessary NVM writes and random access in persistent data structures.

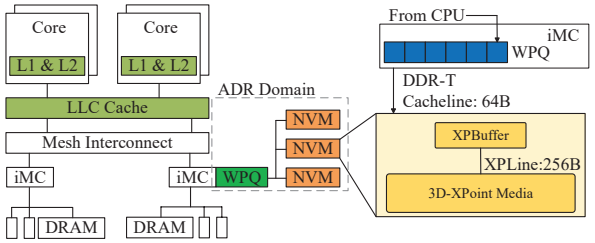
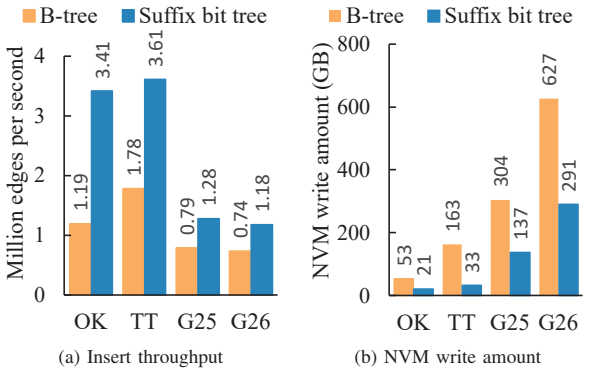


Fig. 1: The architecture of the NVM system and the internal architecture of Intel Optane persistent memory.



(a) Insert throughput (b) NVM write amount

Fig. 2: Comparison between the B-tree and the suffix bit tree.

System architecture. Figure 1 shows the architecture of the NVM system and the internal architecture of Optane persistent memory. The persistent memory is attached to the memory bus, which sits one level below the CPU's multi-level caches. Updates are usually first sent to CPU caches, and for safe persistence, the persistent memory (the first-generation Optane device) requires software to actively use cache line flush instructions, such as clwb. Upon flush, the data will reach the memory controller's write pending queues (WPQ), which is included in the asynchronous DRAM refresh (ADR) domain, and survive power failures. To guarantee specific ordering of writes, applications should also issue sfence to prevent stores from being re-ordered by the CPU. However, using special instructions (clwb and sfence) to persist data from the CPU cache to NVM can drastically slow down write performance by up to an order of magnitude. Therefore, it is important to reduce persistent operations in persistent data structures.

B. Challenges for Dynamic Graphs on NVM

Higher write amplification overhead. Almost all in-memory graph processing systems are write-intensive, and data structures used in these systems are highly unsuitable for direct porting to NVM. For instance, Terrace [8] proposes a hierarchical design that stores neighbors of medium-degree vertices in a shared packed memory array (PMA) [12] and neighbors of high-degree vertices in per-vertex B-trees. PMA separates an array into several segments and assigns the lower and upper bound density thresholds for each segment. When an insertion causes the density of a segment to exceed the threshold, PMA adjusts the density by redispatching all elements stored in the segment's parent, resulting in a higher number of NVM writes, known as write amplification. The same situation exists in the B-tree structure. When a node splits, the B-tree copies the intermediate element to its parent, potentially causing splits upwards level by level, and moves all elements on the right of the intermediate element to a new node.

As discussed aforementioned, NVM has more read/write overhead than DRAM and exhibits asymmetric read/write performance. Write amplification in existing graph processing systems will result in worse performance on NVM. To mitigate the impact of higher write amplification on NVM, FTGraph proposes the degree-aware suffix bit tree to support fast graph insertion, achievable in O(1) write count, along with a compact edge data representation for algorithm analysis. Figure 2 illustrates the insertion performance and the amount of data written to NVM for both B-Tree and our suffix bit tree when inserting four different datasets. It can be seen that the suffix bit tree writes only 20%-48% of the data amount compared to the B-tree. This discrepancy contributes significantly to the insertion performance bottleneck observed in the B-tree.

Higher crash consistency overhead. NVM is non-volatile, and an unexpected system crash can result in incomplete writes or data loss upon the system reboot. This underscores the importance of ensuring that data updates are either fully completed or not initiated at all. Common methods of maintaining data consistency, such as logging or copy-on-write (COW), introduce redundant writes to NVM, resulting in higher write overheads. In addition, the CPU initially writes data to caches, and then leverages explicit cache line flush instructions and memory barriers to guarantee the persistence of NVM writes. The lower write bandwidth of NVM compared to DRAM, along with added data persistence instructions, contributes to the increased write overhead of NVM. Given the CPU's guarantee of 8-byte granularity for atomic writes, our approach in FTGraph capitalizes on this feature to design a log-free crash consistency guarantee method utilizing 8-byte NVM atomic writes [13].

Higher concurrency control overhead. A single thread struggles to take full advantage of the bandwidth offered by NVM, and encounters challenges in meeting the demanding high throughput requirements of dynamic graphs. Therefore, efficient and correct concurrency control of graph data structures is essential. GraphTinker [4] does not incorporate concurrency control within its data structure. Instead, it achieves parallelization by generating multiple C++ instances. The dataset is partitioned based on the source vertex IDs of the edges, and each interval is then loaded into a separate instance of GraphTinker. This can lead to redundant storage of certain data in every instance, such as vertex property arrays and scatter-gather hashing (SGH), ultimately resulting in significant memory wastage. In graph data structures, where multiple threads may access the same edge block at the same

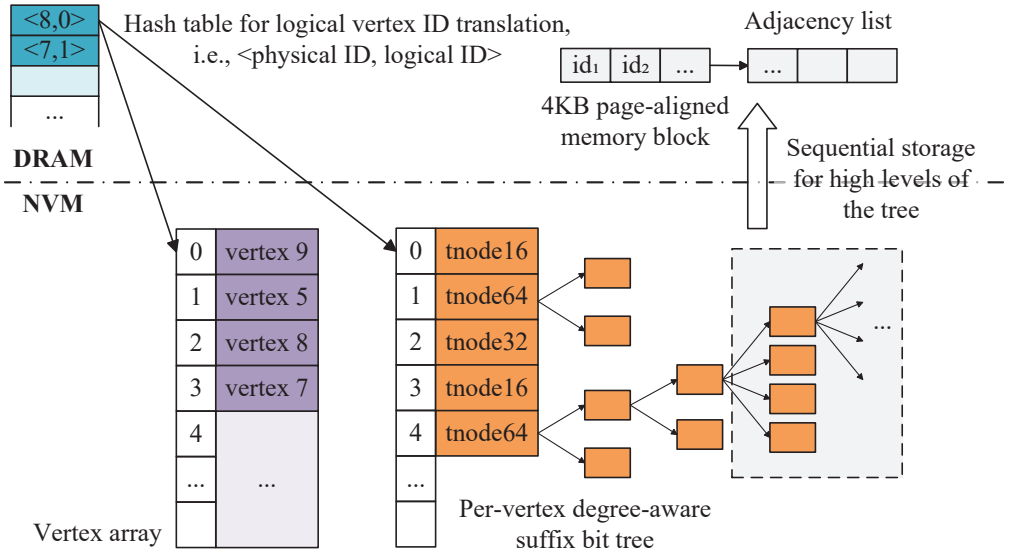


Fig. 3: Overall architecture of FTGraph.

time, a common approach is to place a lock within each edge block to achieve concurrency control. However, the coarse-grained lock mechanism results in greater thread-waiting overhead compared to DRAM, given the higher read/write latency of NVM. Additionally, persistent instructions consume significantly more CPU cycles than normal instructions. To enhance concurrency, FTGraph realizes optimistic version-based concurrency control.

III. FTGRAPH DESIGN

In this section, we first introduce the main idea and the overview of FTGraph, which is a NVM-based graph storage system for large-scale dynamic graphs. Then, we detail the design and the operations for degree-aware suffix bit trees in FTGraph. Finally, we describe how to integrate NVM atomic writes with a fine-grained concurrency mechanism to ensure crash consistency and improve concurrency acceleration.

A. Overview of FTGraph

Graph structure. The overall graph structure of FTGraph is shown in Figure 3. FTGraph's major structures are the in-memory hash table, the in-memory adjacency lists, the vertex array, and the degree-aware suffix bit trees for each vertex, where the latter two are stored on NVM. The vertex array is implemented as a pre-allocated array with a fixed capacity on NVM. This solution ensures quick vertex look-ups, in O(1). In the vertex array, vertices are stored according to the order of their arrival in a semi-dense fashion. To support arbitrary vertex IDs, the vertex array needs to be associated with a hash table. We first check if the two vertex IDs have been hashed before inserting each new edge. If not, the new vertex ID is mapped to the next unused index in the vertex array. Thus each physical vertex id maps to an index in the vertex array by the hash table [4, 2, 3]. The adjacent edges of each vertex are stored within a degree-aware suffix bit tree and each tree node (tnode) contains multiple edges, which will be described in Section III-B in detail. The in-memory hash table index also maps each vertex to the physical position of its first node of the tree. It's effective that vertex look-ups and the retrieval of their first adjacent edges can exhibit a constant cost [2].

Logical vertex ID translation. Furthermore, translating physical vertex IDs to logical indices provides a highly compacted graph data representation. Frequently, most graph algorithms need to scan neighborhoods, i.e., the Bellman-Ford algorithm requires scanning the neighborhoods of $|V|$ vertices in $|V|-1$ iterations, where $|V|$ is the number of vertices in the graph. In fact, there are many empty vertices that have no in-edges and out-edges, and they do not need to be traversed while running the analytics. For example, the Twitter graph has 35% empty vertices. So, translating a physical vertex ID to a logical index helps to reduce the number of iterations and the number of vertices to be traversed in each iteration, improving the analytics performance.

B. Degree-Aware Suffix Bit Tree

The suffix bit tree design. We store the adjacent edges of each vertex in separate suffix bit trees. Figure 4 gives a simple example of a suffix bit tree. As shown, each tree node stores multiple edges and may have two child nodes. The path to its left child is represented as "0", while the path to its right child is represented as "1". The "0" or "1" is a certain bit of a number, which provides fast decision-making for traversal paths. In detail, when inserting or searching an edge, first check the 0th bit of the destination vertex ID value. If it is equal to 0 then continue to insert or search in the left child node at the first level, otherwise continue in the right child node. Then scan the adjacent edges within the tree node to

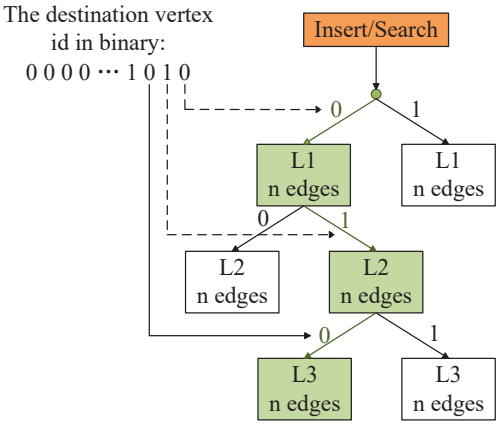


Fig. 4: The suffix bits of the destination vertex ID indicate the path (marked in green) of edge insertion and look-ups in the tree.

insert the target edge at an empty location or search for the target edge. If the insert or search is still unsuccessful after all the entry is inspected, then check the 1st bit of the destination vertex ID of the edge and the same process continues in its child node (if available). Finally, the insert or search is successful at the third level as shown in Figure 4.

This hierarchical representation is effective for updates because it's of the order O(log(n)) to check if the edge to be inserted already exists. Furthermore, compared to the traditional B+ tree, the suffix bit tree keeps the elements in the tree nodes unordered and avoids complex tree node splits/merges. In addition, balancing the suffix bit tree can be achieved without any overhead, as the generally unpredictable destination vertex ID values will cause the suffix bit tree to branch out in a naturally balanced manner. Consequently, the suffix bit tree reduces a lot of unnecessary writes and it's more lightweight and more suitable for NVM.

The degree-aware structure design. The natural graphs commonly found in the real world, like social networks and the web, often exhibit highly skewed power-law degree distributions. Therefore, we evaluated four real-world graphs to analyze their degree characteristics, as shown in Figure 5. In datasets such as Orkut [14] and Twitter [15], vertices with degrees ranging from 1 to 16 can constitute between 20% to 86%, and from 1 to 64 can account for 63% to 97%, respectively. Furthermore, vertices with degrees ranging from 1 to 960 can make up to 99% in almost all datasets. Based on this finding, we target to realize differentiated suffix bit tree structures for different vertices with varying degrees. To achieve the goal, a common approach is adaptively hierarchical data structure representation [16]. Such representations often have multiple levels and initially, there is only one level (L0) with a small size. When the level of L0 is full, a new level (L1) with double the size of L0 is created, and then move all elements to its L1. After that, the new coming element updates would be directly written to its L1 level. When the level of L1 is full, the same process is adopted. Finally, as the size of the new level gets larger and larger, it will limit performance and scalability due to more and more extra data movement overhead across different levels, so special handling is required.

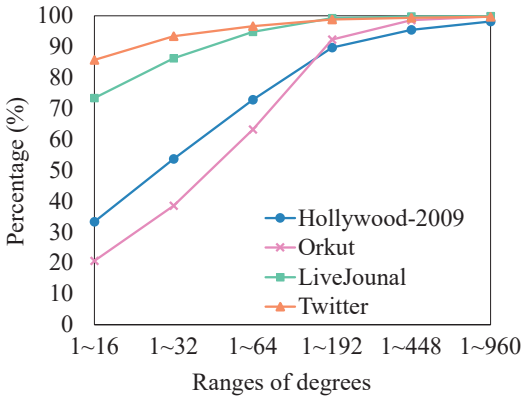


Fig. 5: Percentage of vertices with various ranges of degrees.

Inspired by this, we combine the suffix bit tree with the degree-aware structure design, broadly sketched in Figure 6. Initially, there is only one root node, which can hold at most 16 neighboring edges, i.e. tnode16, as vertices with degrees ranging from 1 to 16 can account for up to 86% in real-world graphs. When the root node with the size of 16 is full, it scales up to a double-size, i.e. tnode32, which can hold at most 32 neighboring edges. Similarly, the root node will finally scale up to a maximum size of 64, i.e. tnode64, as vertices with degrees in the range of 1 to 64 can account for up to 97% in real-world graphs. When there is only a root node, all neighbors of a vertex are stored in consecutive memory, like in an array. Thus, either updates or scans will have a good cache locality.

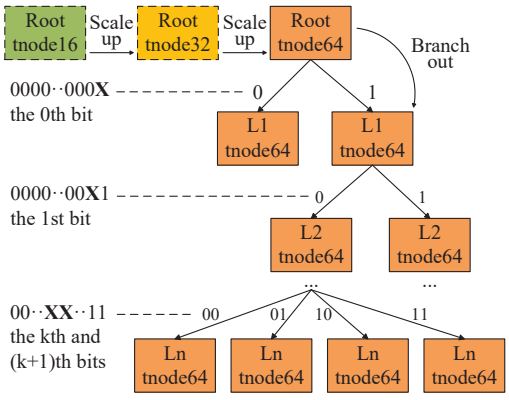


Fig. 6: Overall architecture of degree-aware suffix bit tree with n+1 levels.

After scaling up to the maximum size of 64, the root node starts to branch out like the aforementioned suffix bit tree when it's full. First, we create two children of the root node with the

same size as the first level of the suffix bit tree in a contiguous part of NVM. The 0th bit of the destination vertex ID is then checked to determine the insert path.

However, if one bit is checked at each level while inserting edges, the suffix bit tree will have a large height and need a lot of levels of chaining for the hub vertices, thus increasing the number of accesses to NVM. To mitigate this issue, two bits are checked together to determine the insert path after the levels of the suffix bit tree exceed a certain threshold. In this way, a full tree node will branch out in the manner of a quadtree. As vertices with degrees in the range of 1 to 960 can account for up to 99% in almost all datasets, we set the threshold to 4 (including the level of the root node, i.e. L0), so the first 4 levels of the suffix bit tree can hold at most 960 neighboring edges (15 tree nodes $\times$ 64 edges per tree node = 960 edges), which exploits the structure of power-law graphs.

The tree node design. Figure 7 shows the data structure of a degree-aware suffix bit tree node with 64 edges. Each row represents a cache line and is all cache line aligned for efficient cache line accesses [17, 18]. The first cache line stores auxiliary data, including next (the pointer to the next level), locate bitmap (locating empty positions), set bitmap (updating empty positions), level (the tree node level, starting from 0), size (the number of edges a tree node can hold), version (the tree node version). One bitmap takes 8B, which is used to achieve write atomicity, and each bit of which corresponds to an edge position. "1" means that the position is occupied by a valid edge. "0" means that the position is empty. We use the dual bitmap design to integrate 8B NVM atomic writes with optimistic version-based concurrency control, which we introduce later.

The second cache line stores the fingerprint array. The destination vertex id value is hashed to a 1B fingerprint when inserting an edge and then the fingerprint is inserted into the fingerprint array in the same position as the edge. The typical bitmap+fingerprint node design is widely used on NVM indexes [19, 17, 20, 21], which can improve search computation inside a tree node with a single SIMD instruction. Three sizes of tree nodes can store up to 16, 32, and 64 edges, respectively. We accelerate the fingerprint array matching using a single 128-bit (sse2), 256-bit (avx2), and 512-bit (avx512) SIMD instruction, respectively. Finally, unsorted edges start from the third cache line, sharing the same source vertex.

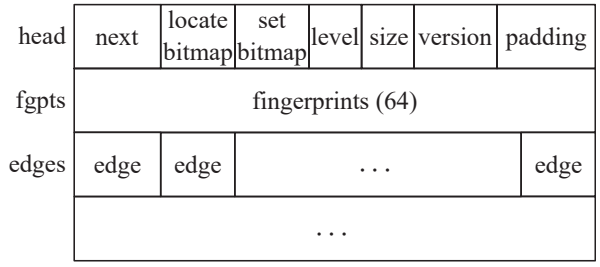


Fig. 7: Data structure of a tree node with 64 edges.

Sequential storage for high levels of the tree. The tree structure does not provide a highly compacted representation of edge data, especially quadtrees. When an edge scan iterates over all nodes of a tree, it needs random accesses, resulting in poor data locality. Moreover, NVM delivers about 3$\times$ higher latency for random accesses than sequential accesses for both reads and writes [17], further increasing the cost of random memory scans. To support efficient data access for graph analytics, FTGraph manages the copy of high levels of each tree in separate per-vertex DRAM-based edge lists. We allocate multiples of 4KB page-aligned memory for these edge lists, and each memory block uses a pointer to the next block. As described, only 1% of vertices have degrees greater than 960 in almost all datasets, we set the start level to 4. As a result, when an edge scan reaches the level of L4, it iterates over the edge list in DRAM instead, avoiding frequent memory jumps in quadtrees. Note that, the copy strategy incurs extra data movement overhead and DRAM space usage, but this overhead is acceptable because there are only 1% of vertices need extra edge lists.

C. Crash Consistency and Optimistic Concurrency

With the FTGraph design, there are two bitmaps stored inside each tree node. The locate bitmap is only used for a thread to locate an empty position before the insert, while the corresponding bit in the set bitmap is updated to make the new edge visible after the insert. Each bitmap can be updated with a single 8-byte failure-atomic write instruction, saving the data consistency costs. An insert updates the two bitmaps successively. When the two bitmaps are not the same, it indicates that there is a writer inside the tree node and a thread (writer or reader) should wait until the two bitmaps are the same. Then a reader optimistically performs an operation and finally checks the locate bitmap again. If two versions of the locate bitmap vary, the operation is retried. Thus, the dual bitmap is also a natural version lock, which allows multiple readers or one writer to access the node at the same time. In this way, we can integrate a log-free crash consistency guarantee approach with optimistic version-based concurrency control through a dual bitmap design.

Log-free and concurrent insert. The insertion operation inside a tree node consists of following steps: (1) A thread first reads the two bitmaps and checks if there is no writer (i.e., the two bitmaps are the same). If not, the thread should wait until the two bitmaps are the same. (2) The thread then checks if an edge already exists. If so, the insert ends. (3) The thread compares the locate bitmap read in step (1) with the current locate bitmap. If two versions vary, the thread should restart from step (1). If not, the thread finds the first empty position in the locate bitmap and sets the corresponding bit to 1. Notably, the compare-and-update operation should be done by a failure-atomic CAS (compare-and-swap) instruction. (4) The new edge is inserted into the corresponding position and persisted to NVM via a sequence of CLWB and SFENCE instructions. Next, the 1-byte hash value of the destination vertex ID is also inserted into the corresponding position in

the fingerprint array, and the second cache line of the tree node is persisted. (5) The thread sets the corresponding bit in the set bitmap via another CAS instruction. Since the locate bitmap and the set bitmap are in the same cache line, i.e. the first cache line of the tree node, we only need one persistence operation. Once the cache line has been flushed successfully, the insert is considered crash-consistent. Because we use the set bitmap to indicate the validity of the edges, a partially written edge remains invalid until the set bitmap is successfully flushed to NVM, thus ensuring data consistency of the insert in the event of a system crash. In addition, each thread gets its specific empty position in the locate bitmap, and once the position is occupied, each thread can execute the insert and slow persistent instructions outside the critical section without interfering with each other, improving the scalability and the graph update performance.

Optimistic version-based concurrent search. A search operation involves traveling the levels downwards and finding the target edge inside the tree nodes. Therefore, there is find-insert contention inside the tree nodes and travel-scale-up/branch-out contention outside the tree nodes. While the dual bitmap can be considered to be a version lock, it's easy to solve find-insert contention inside the tree nodes as described. Similar to PACTree [21], we put an extra optimistic persistent version lock in each tree node to solve travel-scale-up/branch-out contention. A scale-up or branch-out atomically increments the version number upon lock and unlock and the version number becomes odd and even, respectively. A travel (jumping to the next level) first checks if there is no scale-up or branch-out (i.e., an even version number). If so, the travel starts and then checks the version number again. If two numbers vary, the travel will retry, thus solving travel-scale-up/branch-out contention outside the tree nodes.

Log-free scale-ups and branch-outs. When the root nodes with size 16 or 32 become congested and filled with edges, they scale up into the root nodes with size 32 or 64. To avoid tree node space allocation and release and element copying when the root nodes scale up, we just allocate the same size of NVM space to the root nodes with a size of 16 as we do to the root nodes with a size of 64 (trading space for time). The auxiliary parameter size is used to indicate the size of the current tree node. When the root nodes with size 16 or 32 scale up into the root nodes with size 32 or 64, the parameter size is set to 32 or 64, which not only reduces NVM writes and NVM memory management overhead but also facilitates maintaining crash consistency. We set the parameter size to 32 or 64 to achieve scale-ups through an 8-byte NVM atomic write, followed by a sequence of CLWB and SFENCE instructions. When the tree nodes with a size of 64 branch out, their 2 or 4 child nodes are allocated simultaneously in a contiguous segment of NVM memory space. The auxiliary parameter next is used to point to the start address of the space of the child nodes. Similarly, the parameter next is updated with a single 8-byte failure-atomic write followed by persistent instructions. Thus the next is either set successfully via an NVM atomic write or remains at its initial value 0.

IV. EVALUATION

A. Experimental Setup

Machine Configuration. We conducted our experiments on a dual-socket machine equipped with two Intel Xeon Gold 5218 @ 2.3GHz (16 cores / 32 threads in SMT mode), 256GB DRAM, and 1 TB Intel Optane persistent memory 100 Series in App Direct mode (four DIMMs of 128GB per socket). The server runs Ubuntu 20.04.1 LTS with a Linux kernel of 5.15.0 and the Ext4-Dax file system is mounted on the PM devices. Due to the NUMA effects, we bind our programs to run on socket 1 (using pmem1), thus avoiding remote NVM reads and writes across the NUMA nodes.

Compared Systems. We considered the following compared systems :

- **GraphTinker** [4]: an in-memory data structure for dynamic graphs, which combines the benefits of Robin Hood and Tree-Based Hashing to reduce the probe distance and improve the update throughput. We implement GraphTinker-N on NVM based on GraphTinker, and we only store the whole EdgeblockArray on NVM and keep the vertex property array, the scatter-gather hashing unit, and the coarse adjacency list (CAL) in DRAM.
- **Terrace** [8]: an in-memory data structure for dynamic graphs, which adopts a hierarchical data structure design by taking advantage of the inherent skewness in the degree distribution of real-world graphs. We implement Terrace-N on NVM based on Terrace, and we only store the in-place (first) level on NVM and keep the second level and third level on DRAM.
- **XPGraph** [16]: an XPLine-friendly NVM-based graph storage system, which allocates an adjacency list in DRAM to cache edge updates for each vertex and flushes the entire buffer to NVM when it is full, thus merging multiple XPLine accesses to NVM into single XPLine access.

Graph datasets. We use both real-world graphs and synthetic graphs in our experiments. Orkut [14] and Twitter [15] are real-world graphs, while Graph500-25 and Graph500-26 are generated using graph500 generator [22]. All graphs have skewed power-law degree distributions and are widely used in other graph system evaluations. The detailed of these graph datasets is shown in Table I. Note that, we represent undirected edges a-b as two directed edges: a→b, b→a [2].

TABLE I: Graph datasets with number of vertices, number of edges, and average degree of vertices

Graphs	Vertices $\|V\|$	Edges $\|E\|$	D
Orkut (OK)	3.0×10^6	234×10^6	76
Twitter (TT)	61.6×10^6	1.4×10^9	23
Graph500-25 (G25)	33.5×10^6	1.0×10^9	31
Graph500-26 (G26)	67.1×10^6	2.1×10^9	31

Evaluation metrics. We first evaluate FTGraph and its competitors in graph update performance, which includes single-threaded insert and multi-threaded insert. Then we

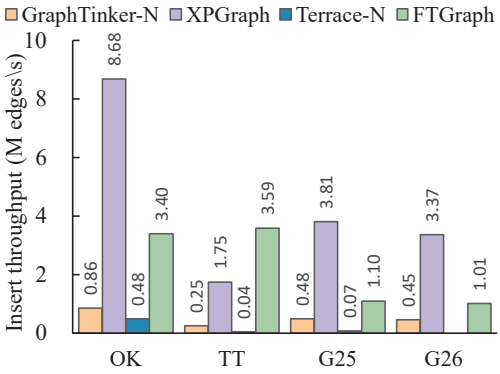


Fig. 8: Graph insertion time among various graph systems on different datasets and with a batch size of one million edges.

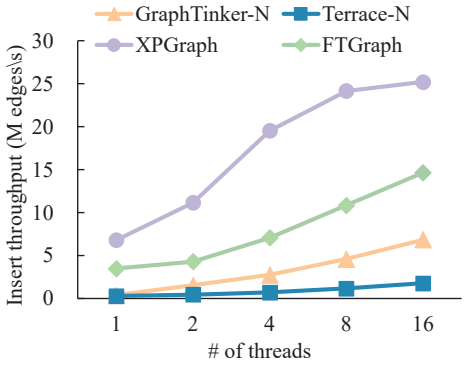


Fig. 9: Multi-threaded graph insertion throughput among various graph systems on the Orkut dataset and with a batch size of one million edges.

choose four algorithms, including BFS (breadth-first-search), CC (connected-components), PR (page-rank), SSSP (single-source-shortest-path), and a 2-Hop query to simulate the various use cases to demonstrate the impact of FTGraph on analytics. We verify the correctness of multi-threaded insert by the fact that every edge in the datasets can be found. The algorithms are almost exactly the same as in XPGraph. The results of the algorithms and the queries executed by different graph systems should be the same so we verify the correctness of the algorithms and the queries by the same outputs.

B. Graph Update Performance

Single-threaded insertion. The experiment starts with an empty data structure and imports the graph data in batches. The batch size of edges used in our experiments is 1 million edges per batch. Figure 8 shows the throughput of various graphs for single-threaded inserts using different datasets. Missing bars indicate that a system could not load the graph due to memory restrictions. FTGraph outperforms Terrace-N and GraphTinker-N across all the datasets, and, conversely, Terrace-N is always slower than others. More precisely, FTGraph is consistently faster and achieves 1.9× to 3.8× speedups than GraphTinker-N and 5.6× to 21.2× speedups than Terrace-N for different graphs.

Moreover, we also find that XPGraph is always faster than others on various datasets except Twitter, there are three main reasons for this. First, XPGraph does not perform the check if an edge exists, which helps to improve the insert performance but may lead to duplicate edges and wrong results of graph analysis. Second, XPGraph has high hidden costs, as it uses background threads to perform the buffering phase and the flushing phase. Third, XPGraph uses DRAM as a cache to batch updates into the adjacency lists, which amortizes the NVM access cost for each edge update.

The three other systems all check if an edge exists. FTGraph is superior to the others because the flexible suffix bit trees allow for efficient checks. Inside the tree nodes, FTGraph uses SIMD instructions to accelerate the process of checking whether any fingerprint matches the search edge's fingerprint. It then only accesses entries with matching fingerprints, skipping all the other entries. Outside the tree nodes, FTGraph travels to the target tree node according to the source vertex ID's suffix bits with the order O(logN), where N is the number of tree nodes. Furthermore, when inserting an edge, FTGraph can complete the insertion in O(1) write count after it reaches the target tree node.

Compared to others, FTGraph has almost no write amplification. GraphTinker can make fewer edge traverses during updates through Robin Hood Hashing and Tree-Based Hashing. However, when a new edge is inserted, the Robin Hood Hashing may cause more edges to move. In addition, GraphTinker maintains a separate CAL representation, and every edge needs to be copied again into the CAL, thus increasing maintenance overhead. Terrace introduces a hierarchical design and mainly stores neighbors in a single shared PMA (the second level) and per-vertex B-Trees (the third level). Both the PMA and B-trees cause high write amplification as described. In addition, if the degree of a vertex becomes greater than the maximum number of neighbors that can be stored in the PMA, Terrace removes that vertex's neighbors from the PMA and inserts them in a B-tree along with the new incoming neighbors, which results in serious write amplification and poor insert performance.

Multi-threaded insertion. Figure 9 shows the insertion throughput of FTGraph and other graph systems as the number of threads increases on the Orkut dataset. As shown, XPGraph is superior to the others for similar reasons as single-threaded. FTGraph performs better than the other two in each case, which achieves 2.14× to 8.85× higher throughput than GraphTinker-N, and 8.26× to 12.04× higher throughput than Terrace-N.

GraphTinker realizes the parallelism by partitioning the edge dataset into several instances according to their source vertex IDs. It is an ideal approach to concurrency, as there is no read-writer contention among GraphTinker instances. But in terms of scalability, we can observe that FTGraph has linear scalability like GraphTinker, as FTGraph realizes a relaxed and fine-grained concurrency control mechanism to

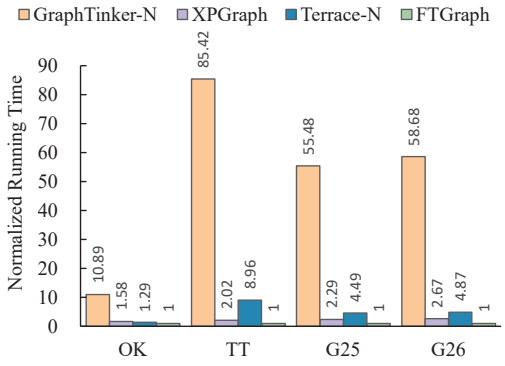


Fig. 10: Graph query performance among various graph systems using different datasets when running the 2-HOP query.

reduce the read-writer contention. In addition, FTGraph has a higher throughput overall. For the same reason as in the single-threaded case, high write amplification in edge blocks and the CAL representation slow down the rate of inserts. Terrace always suffers from the high cost of deleting a vertex's neighbors from the PMA and inserting them in a B-tree.

Memory usage. Table II reports the memory footprint of the different systems when running on the Twitter dataset. FTGraph uses the least amount of DRAM, as it only requires DRAM to store the high levels of each tree for fast graph analysis. FTGraph's overhead of NVM mainly comes from the pre-allocated vertex array and root nodes of each tree due to many empty vertices. On the other hand, FTGraph allocates two or four new tree nodes simultaneously when a tree node branches out. This also leads to higher space usage.

TABLE II: Memory footprint (in GB) of different systems when running on the Twitter graph.

Media	GraphTinker-N	XPGraph	Terrace-N	FTGraph
DRAM	119.86	4.13	26.13	2.22
NVM	267.33	90.90	3.67	35.15

C. Graph Analysis Performance

In this section, we first evaluate graph query performance among various graphs, i.e., the 2-HOP neighbor query, and then evaluate graph computation performance via four common graph algorithms BFS, CC, PR, and BF. We execute the same algorithm implementations on all systems evaluated.

For the 2-HOP query, we first access the neighbors of random 2^{15} non-zero degree vertices and then access the neighbors of the corresponding 1-HOP neighbors. The different graph systems access the neighbors of the same selected vertices to make sure the complexity of the query is the same. As shown in Figure 10, FTGraph performs better than other graph systems when running the 2-HOP query.

As for graph analytic algorithms BFS, CC, PR, and BF, FTGraph always achieves the best performance. From Figure 11, it can been seen that compared to GraphTinker-N, FTGraph achieves up to 5.56×, 6.84×, 7.54×, and 6.31× speedup for BFS, CC, PR, and BF respectively. Accordingly, FTGraph achieves up to 2.60×, 5.92×, 2.96×, and 3.24× better speedup than Terrace-N. For XPGraph, we also observe that FTGraph performs up to 4.83× better than it.

We believe there are three major causes for that. First, real-world graphs generally have a power-law distribution, and only a small part of vertices have degrees larger than 64 as noted in Figure 5, so the neighbors of most vertices can be stored in just one root node of size 64, or even in root nodes of size 32 and 16. For these low-degree vertices, edges are stored sequentially in a contiguous tree node, which benefits fast scans for graph analysis. Second, as all children of a tree node are allocated simultaneously in the same contiguous segment of NVM, the hierarchical traversal of the trees provides an opportunity for contiguous child node accesses and good locality. Further, we add DRAM-based edge lists for compacted edge representation of high levels of the tree, which offers a good data access locality and significantly reduces the number of random access on NVM. Third, logical vertex ID translation provides a highly compacted graph data representation and skips the computations for empty vertices.

GraphTinker-N is almost always slower than the other systems. It maintains a separate copy of the edges in the database by a coarse adjacency list (CAL) representation, achieving a more compact edge data representation in the database but suffering from traveling more edges. Because several source vertices share an adjacency list in the CAL representation, the edges in an adjacency list can belong to different vertices. When retrieving the edges of a vertex, all edges of other vertices are also traveled, which may incur a greater latency.

Both Terrace-N and XPGraph do not use a logical vertex ID translation strategy. As a result, they suffer from a lot of unnecessary traverses for empty vertices. In addition, we can observe that Terrace-N performs poorly when running on large graphs, such as Twitter, Graph500-25, and Graph500-26. Terrace-N adopts individual B-trees for each high-degree vertex, and each B-tree scatters the neighbors into multiple tree nodes, which does not feature efficient scans due to many random accesses.

D. Efficiency of edge lists.

First, we measure the impact of edge lists on graph update and analysis performance. Any edge inserted into the tree nodes with a level greater than 3 should be copied into a separate edge list. This slows down the performance of graph updates as shown in Table III. There is a 0.52%-9.37% slowdown compared to its original version without edge lists. In summary, adding edge lists has little effect on graph updates because the edge list format supports fast updates as each update is simply appended to the end of the list. As for the impact on graph analysis, we can observe that it has a limited improvement when running a small dataset, i.e., Orkut, as there are a few vertices that have high degrees and need separate edge lists. But FTGraph can run faster on different benchmarks

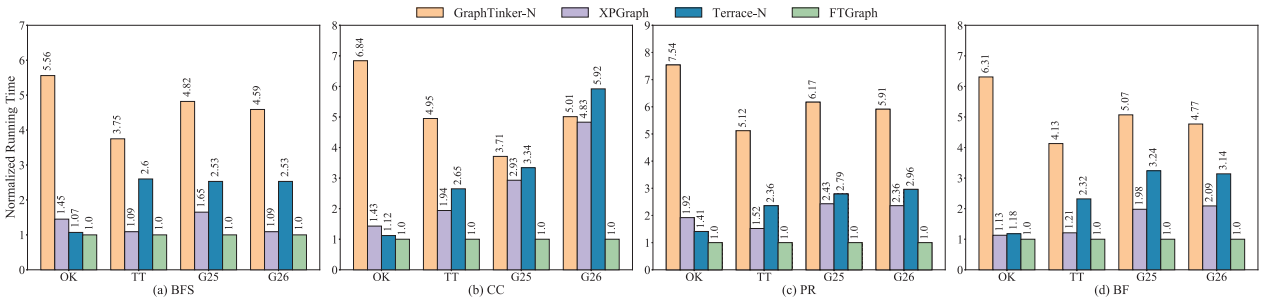


Fig. 11: Comparative analysis of different graph systems using various datasets and algorithms. The normalized running time refers to the performance ratios of various systems compared to FTGraph, with smaller values indicating better efficiency.

when running on other datasets by up to 83%. Intuitively, the sequential storage format of edge lists can significantly reduce the number of random accesses on NVM.

Then, we measure the benefit of edge lists on graph analysis performance when setting a different start level and show the results in Figure 12. It's obvious that the more edges of a tree are copied into the edge list, the higher the graph analysis performance. However, larger edge lists cost more DRAM space. For example, it costs 2x the DRAM space when we set the default start level from 4 to 0. Therefore, the start level of 4 is a good choice to balance DRAM space usage and graph analysis.

TABLE III: Percentage of performance reduction in graph updates and performance improvement in graph analysis when adding edge lists.

Graphs	2-HOP	BFS	PR	CC	BF	Insert
OK	17.96%	0.85%	2.07%	6.36%	0.72%	0.52%
TT	80.98%	5.89%	1.72%	7.17%	7.71%	1.65%
G25	80.90%	16.92%	13.42%	11.90%	18.75%	5.22%
G26	83.31%	14.66%	13.27%	15.01%	16.41%	9.37%

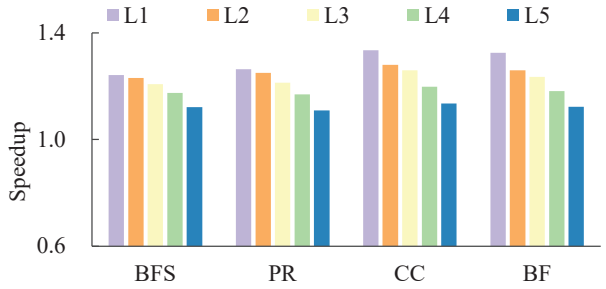


Fig. 12: Speedup for graph analysis on dataset Graph500-26 when adding edge lists from different start levels. L1 denotes adding edge lists for tree nodes with a level $>= 1$ and lower start levels mean more DRAM space is used.

V. RELATED WORK

Large-Scale Dynamic Graph Processing. Dynamic graphs refer to the fact that graph structures are continuously changing at very high rates. Therefore, dynamic graph processing systems often prefer to work in main memory to achieve high performance, scalability, as well as high memory efficiency [4, 3, 23, 24, 6, 2, 5]. However, memory requirements for storing these data can be much larger than DRAM size [25]. Many distributed graph systems have been proposed to overcome this limitation through partitioning and placing the graph data on clusters of machines, but they often encounter challenges due to networking latency [26, 27, 28]. In addition, some graph systems store data on external storage devices (e.g., hard disks or SSDs) [29, 30]. However, the high I/O cost of internal and external memory interaction limits their performance. To address the issues, we propose FTGraph, which exploits NVMs to provide larger storage capacity and superior performance for large-scale dynamic graphs. Notably, there are also some frameworks designed for GPU architectures [31, 32, 33]. These frameworks often target at high update rates and efficient memory management techniques on GPUs.

To the best of our knowledge, the novel suffix bit tree was first proposed by us for graph processing. We believe that the suffix bit tree offers promise for building high-performance dynamic graph representations. There are some existing approaches that use suffix trees to handle graph-related work [34, 35]. However, it is crucial to note that the traditional suffix tree is completely different from the suffix bit tree. The former is often used in text search and information retrieval, where each text or document is associated with a suffix tree, and each edge is labeled with a substring of the text or the document.

Persistent Index. Several techniques applied in FTGraph derive from previous work in the domain of persistent indexes. Earlier studies have proposed various ways to maintain crash consistency, such as logging [36, 37], shadowing [38], PMwCAS [39], and NVM atomic writes [13, 40]. Recent works often target using lightweight NVM atomic writes to avoid serious performance degradation caused by logging and shadowing. The bitmap, which is widely used to implement NVM atomic writes [19, 17, 21, 18, 20, 13], can be updated with a single 8-byte failure-atomic write instruction, saving the data consistency cost. In FTGraph, we also use the bitmap to ensure crash consistency. Moreover, since the read/write latency of NVM is relatively higher and slow persistent instructions in a critical section may block the execution of other threads, special data structures are often required to support effective synchronization. FPTree [20], LB+Tree [19], and RNTree [18] use HTM (hardware transactional memory) for concurrent

control, which allows concurrent read operations to proceed at the same time. More relaxed concurrency access mechanisms, such as read-write lock [17], version lock [21], and lock-free design [39], also allow more operations to be executed concurrently. Distinct from existing methods, we integrate the graph structure with NVM characteristics, utilizing a dual bitmap and NVM atomic writes to achieve crash consistency and optimistic version-based concurrency control.

VI. CONCLUSION

In this paper, we proposed FTGraph, a persistent memory data structure for large-scale dynamic graph processing, which highlights the update performance of dynamic graphs while simultaneously supporting high-performance queries and analytics. In FTGraph, neighborhoods of each vertex are stored in separate per-vertex degree-aware suffix bit trees, leveraging the structure of power-law graphs to reduce the edge probe distance and the number of NVM writes when updating an edge. FTGraph integrates the design of consistency and concurrency through the dual bitmap, avoiding redundant writes and reducing the potential synchronization overhead in NVM. Results show that FTGraph outperforms state-of-the-art graph processing systems by up to 21.2× in graph updates and 85.4× in graph analysis. The next steps of FTGraph include reducing the DRAM/NVM space usage and exploring a hybrid DRAM-NVM approach for both graph updates and graph analysis.

ACKNOWLEDGMENT

We are thankful to the anonymous reviewers for their valuable feedback. This work is supported in part by the National Science Foundation of China (62372450), the National Key Research and Development Program of China (2021ZD0110700), the National Science Foundation of China (62172361), and the Major Projects of Zhejiang Province (LD24F020012).

REFERENCES

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

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

[3] D. Ediger, R. McColl, J. Riedy, and D. A. Bader, "STINGER: High performance data structure for streaming graphs," in *the IEEE Conference on High Performance Extreme Computing*. IEEE, 2012, pp. 1–5.

[4] W. Jaiyeoba and K. Skadron, "GraphTinker: A high performance data structure for dynamic graph processing," in *the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2019, pp. 1030–1041.

[5] P. Kumar and H. H. Huang, "GraphOne: A data store for real-time analytics on evolving graphs," *ACM Transactions on Storage (TOS)*, vol. 15, no. 4, pp. 1–40, 2020.

[6] X. Zhu, G. Feng, M. Serafini, X. Ma, J. Yu, L. Xie, A. Aboulnaga, and W. Chen, "LiveGraph: A transactional graph storage system with purely sequential adjacency list scans," *Proceedings of the VLDB Endowment*, vol. 13, no. 7, pp. 1020–1034, 2020.

[7] P. Celis, P. A. Larson, and J. I. Munro, "Robin hood hashing," in *the 26th Annual Symposium on Foundations of Computer Science*, 1985.

[8] P. Pandey, B. Wheatman, H. Xu, and A. Buluc, "Terrace: A hierarchical graph container for skewed dynamic graphs," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 1372–1385.

[9] (2019) Intel Optane DC persistent memory architecture overview. [Online]. Available: https://www.intel.com/content/www/us/en/products/details/memory-storage/optane-dc-persistent-memory.html.

[10] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An empirical guide to the behavior and use of scalable persistent memory," in *Proceedings of the USENIX Conference on File and Storage Technologies*, 2020, pp. 169–182.

[11] K. Huang, Y. He, and T. Wang, "The past, present and future of indexing on persistent memory," *Proceedings of the VLDB Endowment*, vol. 15, no. 12, pp. 3774–3777, 2022.

[12] M. A. Bender and H. Hu, "An adaptive packed-memory array," *ACM Transactions on Database Systems (TODS)*, vol. 32, no. 4, pp. 26–es, 2007.

[13] S. Chen and Q. Jin, "Persistent B+-Trees in non-volatile main memory," *Proceedings of the VLDB Endowment*, vol. 8, no. 7, pp. 786–797, 2015.

[14] (2012) Orkut dataset. [Online]. Available: https://snap.stanford.edu/data/com-Orkut.html.

[15] (2010) Twitter dataset. [Online]. Available: https://github.com/ANLAB-KAIST/traces/releases/tag/twitter_rv.net.

[16] R. Wang, S. He, W. Zong, Y. Li, and Y. Xu, "XPGraph: XPline-friendly persistent memory graph stores for large-scale evolving graphs," in *the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 1308–1325.

[17] Z. Li, B. Jiao, S. He, and W. Yu, "PHAST: Hierarchical concurrent log-free skip list for persistent memory," *Transactions on Parallel and Distributed Systems (TPDS)*, vol. 33, no. 12, pp. 3929–3941, 2022.

[18] M. Liu, J. Xing, K. Chen, and Y. Wu, "Building scalable NVM-based B+ tree with HTM," in *Proceedings of the 48th International Conference on Parallel Processing*, 2019, pp. 1–10.

[19] J. Liu, S. Chen, and L. Wang, "LB+-trees: optimizing persistent index performance on 3DXPoint memory," *Proceedings of the VLDB Endowment*, vol. 13, no. 7, pp. 1078–1090, 2020.

[20] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner, "FPTree: A hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory," in *Proceedings of the International Conference on Management*

of Data, 2016, pp. 371–386.

[21] W.-H. Kim, R. M. Krishnan, X. Fu, S. Kashyap, and C. Min, "PACTree: A high performance persistent range index using PAC guidelines," in *Proceedings of the 28th Symposium on Operating Systems Principles (SOSP)*, 2021, pp. 424–439.

[22] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the graph 500," *Cray Users Group (CUG)*, vol. 19, pp. 45–74, 2010.

[23] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2013, pp. 135–146.

[24] N. Sundaram, N. R. Satish, M. M. A. Patwary, S. R. Dulloor, S. G. Vadlamudi, D. Das, and P. Dubey, "GraphMat: High performance graph analytics made productive," *Proceedings of the VLDB Endowment*, vol. 8, no. 11, pp. 1214–1225, 2015.

[25] S. Lim, T. Coy, Z. Lu, B. Ren, and X. Zhang, "NV-Graph: Enforcing crash consistency of evolving network analytics in NVMM systems," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 31, no. 6, pp. 1255–1269, 2020.

[26] R. Chen, J. Shi, Y. Chen, B. Zang, H. Guan, and H. Chen, "PowerLyra: Differentiated graph computation and partitioning on skewed graphs," *ACM Transactions on Parallel Computing (TOPC)*, vol. 5, no. 3, pp. 1–39, 2019.

[27] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed graph-parallel computation on natural graphs," in *the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012, pp. 17–30.

[28] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Distributed graphlab: A framework for machine learning in the cloud," *arXiv preprint arXiv:1204.6078*, 2012.

[29] A. Kyrola, G. Blelloch, and C. Guestrin, "GraphChi: Large-scale graph computation on just a PC," in *the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012, pp. 31–46.

[30] H. Liu and H. H. Huang, "Graphene: Fine-grained IO management for graph computing." in *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST)*, 2017, pp. 285–300.

[31] F. Busato, O. Green, N. Bombieri, and D. A. Bader, "Hornet: An efficient data structure for dynamic sparse graphs and matrices on GPUs," in *the IEEE High Performance extreme Computing Conference (HPEC)*, 2018, pp. 1–7.

[32] M. Winter, D. Mlakar, R. Zayer, H.-P. Seidel, and M. Steinberger, "FaimGraph: high performance management of fully-dynamic graphs under tight memory constraints on the GPU," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2018, pp. 754–766.

[33] D. Sengupta and S. L. Song, "Evograph: On-the-fly efficient mining of evolving graphs on GPU," in *International Conference on High Performance Computing (ISC)*, 2017, pp. 97–119.

[34] M. Lux, S. M. zu Eissen, and M. Granitzer, "Graph retrieval with the suffix tree model," in *the 3rd International Workshop on Text-Based Information Retrieval (TIR-06)*, 2006, p. 30.

[35] V. Bonnici, A. Ferro, R. Giugno, A. Pulvirenti, and D. Shasha, "Enhancing graph database indexing by suffix tree structure," *Pattern Recognition in Bioinformatics*, pp. 195–203, 2010.

[36] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, "ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging," *ACM Transactions on Database Systems (TODS)*, vol. 17, no. 1, pp. 94–162, 1992.

[37] A. A. R. Islam and D. Dai, "DGAP: Efficient dynamic graph analysis on persistent memory," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2023.

[38] 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 Symposium on Operating Systems Principles*, 2009, pp. 133–146.

[39] J. Arulraj, J. Levandoski, U. F. Minhas, and P.-A. Larson, "BzTree: A high-performance latch-free range index for non-volatile memory," *Proceedings of the VLDB Endowment*, vol. 11, no. 5, pp. 553–565, 2018.

[40] Z. Liu and S. Chen, "Pea Hash: A performant extendible adaptive hashing index," *Proceedings of the ACM on Management of Data*, vol. 1, no. 1, pp. 1–25, 2023.