GTX: A Write-Optimized Latch-free Graph Data System with Transactional Support

LIBIN ZHOU, Purdue University, USA
LU XING, Purdue University, USA
YEASIR RAYHAN, Purdue University, USA
WALID G. AREF, Purdue University, USA

This paper introduces GTX, a standalone main-memory write-optimized graph data system that specializes in structural and graph property updates while enabling concurrent reads and graph analytics through ACID transactions. Recent graph systems target concurrent read and write support while guaranteeing transaction semantics. However, their performance suffers from updates with real-world temporal locality over the same vertices and edges due to vertex-centric lock contentions. GTX has an adaptive delta-chain locking protocol on top of a carefully designed latch-free graph storage. It eliminates vertex-level locking contention, and adapts to real-life workloads while maintaining sequential access to the graph's adjacency lists storage. GTX's transactions further support cache-friendly block-level concurrency control, and cooperative group commit and garbage collection. This combination of features ensures high update throughput and provides low latency graph analytics. Based on experimental evaluation, in addition to not sacrificing the performance of read-heavy analytical workloads, and having competitive performance similar to state-of-the-art systems, GTX has high read-write transaction throughput. For write-heavy transactional workloads, GTX achieves up to 11× better transaction throughput than the best-performing state-of-the-art system.

CCS Concepts: • **Information systems** → **Graph-based database models**; **Data management systems**; **Data locking**; *Main memory engines.*

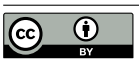
Additional Key Words and Phrases: Dynamic graph, transaction, latch-free concurrency control

ACM Reference Format:
Libin Zhou, Lu Xing, Yeasir Rayhan, and Walid G. Aref. 2025. GTX: A Write-Optimized Latch-free Graph Data System with Transactional Support. *Proc. ACM Manag. Data* 3, 3 (SIGMOD), Article 168 (June 2025), 27 pages. https://doi.org/10.1145/3725305

1 Introduction

Managing and querying dynamic graphs have become increasingly important in many real-world applications, e.g., recommendation systems, fraud detection, threat detection, geo-spatial navigation, e-commerce, knowledge graph applications, and risk management [10, 17, 18, 55, 62, 72, 86, 89, 102]. Graphs continuously change [3, 5, 21, 23, 27, 30, 53, 72, 84, 90] with millions of edge updates per second [27], e.g., 500 million tweets generated in Twitter per day [21]. Applications require transactions to update multiple vertices and edges [72], e.g., in ByteDance services [8], when a user creates an article, a transaction atomically inserts 3 edges: (user, article), (user, tag), (article, tag). Without transactions, updates can corrupt the data [35], e.g., violating the reciprocal consistency that requires atomically updating two edges between a pair of vertices [35, 101]. At the same time,

Authors' Contact Information: Libin Zhou, zhou822@purdue.edu, Purdue University, West Lafayette, Indiana, USA; Lu Xing, Purdue University, West Lafayette, Indiana, USA, xingl@purdue.edu; Yeasir Rayhan, Purdue University, West Lafayette, Indiana, USA, yrayhan@purdue.edu; Walid G. Aref, Purdue University, West Lafayette, Indiana, USA, aref@purdue.edu.



ACM 2836-6573/2025/6-ART168
https://doi.org/10.1145/3725305

dynamic graphs need to support graph analytics [36, 39, 52, 72, 111]. ByteDance [8] detects fraud, and manages risk via subgraph pattern-matching that may traverse multiple hops while the graph is being updated concurrently [72]. Without transactional atomicity, consistency, and isolation, graph analytics algorithms, originally designed for static graphs, cannot be run [52], and may yield incorrect results [35, 39] and security vulnerabilities [35, 100], e.g., an access control system needs to update access permissions of roles and entities atomically [35, 100]. Without transactional guarantees, users may obtain incorrect permissions, and malicious users can exploit race conditions to trigger vulnerabilities. Graph systems, e.g., [44, 64, 78], that do not support transactions cannot run graph analytics concurrently with updates [52]. Recent graph data systems, e.g., [39, 52, 111], support mixed transactions and analytics workloads.

Experiments [52] show that the performance of state-of-the-art transactional graph systems [39, 52, 111] suffer significantly when workloads follow real-world update patterns. Many real-world scenarios exhibit power-law graphs with vertex degrees following power-law distribution [33, 34, 46, 95, 111], and super vertices (hub vertices) [52, 72]) having large degree. Also, real-world graph workloads have hotspots, and their edge updates have temporal locality [52, 65]. These update patterns cause not only updates to the same vertex's adjacency list to be temporally close to each other, but also create congestion of large amounts of concurrent updates at the same vertices (i.e., hub vertices becoming hotspots [72]). Vertex-centric locking and the lack of lock-free synchronization cause significant degradation in transaction throughput [52].

Latch-free indexes, e.g., [24, 25, 66, 70, 71, 74, 99, 109], eliminate locks and blocking, and hence offer higher concurrency. They rely on atomic hardware primitives, e.g., *compare_and_swap (CAS)* and *fetch_add* to serialize updates, but by themselves, do not support transactions [70]. The Bw-Tree [70, 99] does not prevent write-write conflicts or provide transaction atomicity and consistency. It requires a separate transaction protocol [42] to handle transactions.

This paper studies realizing a high-throughput transactional graph data system, GTX, using a latch-free graph store and multiversion concurrency control *MVCC* [106]. GTX is a latch-free write-optimized main-memory transactional graph system supporting highly concurrent read-write transactions and graph analytics.

GTX has a cache-friendly latch-free *MVCC* dynamic graph storage managed and accessed completely by non-blocking atomic primitives. It combines pointer-based and sequential delta-chain storage for efficient edge lookup and adjacency list scans. It uses vector-based delta-chains indexes to locate target edges efficiently while preserving cache-effective sequential adjacency lists. GTX's storage is optimized for power-law edge distributions and hub vertices that are prominent in real-world graphs [52, 111]. GTX has an efficient transaction protocol for the underlying latch-free data structure and dynamic graph workload. GTX eliminates vertex- and edge-centric locks, and has an adaptive delta-chain locking protocol. GTX adapts to the temporal locality and hotspots of edge updates by increasing concurrency for "hot" vertices with higher memory cost, and reducing concurrency for the "cold" vertices. To exploit the high concurrency provided by latch-free *MVCC* and adaptive concurrency control, GTX has a low-latency hybrid group commit protocol and cooperative *MVCC* garbage collection. GTX exhibits a cooperative worker thread design. Except for a single commit manager thread, GTX has no service threads. GTX worker threads install committed transactions' updates, and collect garbage while executing each thread's own transactions.

Experiments demonstrate that GTX achieves up to 2× and 11× higher transaction throughput than the best competitor in random order power-law graph edge insertions and in real-world timestamp-ordered power-law graph edge insertions, respectively. For concurrent transaction and graph analytics workloads, GTX has up to 5.3× higher throughput in edge update transactions for write-heavy workloads, and 3.7× higher throughput for read-write balanced workloads over the best competitor systems. The trade-off is that GTX takes between 0.95× to 2× longer time than

the best performing system to execute most of the concurrent graph analytics, but still performs reasonably well for the majority of the workloads.

The contributions of this paper are as follows. We introduce GTX, a write-optimized transactional graph system with high transaction throughput. GTX has a latch-free graph storage that leverages existing techniques of latch-free indexes, and is optimized for adjacency list storage of dynamic graphs. GTX has a delta chains-based adaptive *MVCC*, combining pointer-based delta-chains storage and sequential storage to provide both efficient single edge lookup and adjacency list scans. GTX has a high-throughput transaction protocol equipped with a hybrid commit protocol, and cooperative garbage collection suited for latch-free multi-versioning storage. GTX concurrency control dynamically adapts to real-world update patterns, temporal localities, and hotspots, and eliminates the need for vertex- or edge-centric locking commonly used in graph systems. Our experiments demonstrate that GTX achieves up to 11x higher read-write transaction throughput while maintaining competitive graph analytics latency concurrently.

The rest of this paper proceeds as follows. Section 2 discusses related work on dynamic graph systems. Section 3 overviews GTX. Sections 4 - 7 present GTX's graph store, transaction operations and transaction management. Section 8 presents an extensive experimental study. Finally, Section 9 concludes the paper.

2 Related Work

Data systems have been developed for graph data management and analytics [1, 2, 6, 11, 15, 18, 27–29, 31, 32, 36, 40, 41, 43, 44, 47–51, 54, 56, 58, 60, 61, 64, 69, 72, 73, 77–80, 83, 85, 88, 92–94, 96, 103–105]. Transactional graph systems [39, 52, 111] execute read-write transactions and analytics on dynamic graphs concurrently.

LiveGraph [111], a main-memory graph data system, supports graph analytics and read-write transactions. LiveGraph adopts an adjacency list-based edge storage model, and uses a vertex array index storing pointers to vertex versions and edge blocks. LiveGraph supports *MVCC*. Read-write transactions acquire exclusive locks on vertices, and create a new version to update a vertex, and append a new update log to update an edge. Readers do not need locks, and directly scan the memory block utilizing transaction timestamps to read the visible versions. LiveGraph stores edge versions of the same source vertex consecutively in the same memory block and enables purely sequential adjacency list scans. LiveGraph's sequential storage of adjacency lists is cache-friendly, reduces random memory access and pointer chasing, and facilitates prefetching [39, 52, 111]. It enables efficient graph analytics. LiveGraph's edge block lacks indexes, and scans the whole block for edge lookups and updates (that need to invalidate old edge versions). Also, LiveGraph's vertex-centric locks block concurrent transactions from writing to the same vertex's adjacency list even if they may be updating different edges. Despite competitive graph analytics performance, LiveGraph suffers from low transaction throughput as shown in our experiments in Section 8. GTX uses a similar adjacency list-based *MVCC* storage model indexed by vertex ID, but overcomes these challenges through 1) A latch-free graph storage (Section 4), 2) A delta-chains-based edge multi-versioning and index within edge-deltas blocks (Sections 4.2, 5.2-5.4), 3) A new high-throughput read-write ACID transaction protocol with delta-chain-based concurrency control (Section 6), 4) A cooperative safe-timestamp-based garbage collection technique during normal transaction operations (Section 7).

Teseo [39] stores graph vertices and their adjacency lists in dynamic large sparse arrays [38] inside leaf nodes of a "fat" B^+-tree to provide both efficient updates and adjacency list reads. Teseo has a clustered sparse index on keys in each sparse array segment to support fast point lookup of vertices and edges, and secondary indexes on vertex locations in the segments to initiate adjacency list scans. Teseo has a hybrid latch per sparse array segment that combines conventional and

optimistic latches to support single-writer multi-reader semantics, and reduces concurrency control overhead in read-intensive workloads. A read-only or read-write transaction can acquire a latch either conventionally or optimistically based on the desired operation and transaction type. Teseo spawns service threads for sparse array and B^+-tree structure modifications. When a segment in a sparse array is full, Teseo requires a service thread to determine a rebalance window of multiple segments in the sparse array, lock them, and redistribute entries among these segments. A sparse array resize either creates a new sparse array as the leaf or splits the leaf into two new sparse arrays. These operations require locking multiple memory blocks exclusively. Transactions that do not conflict with normal operations may conflict in locking neighbor segments that may cause threads to block and stall. Also, multiple threads may compete in locking segments for rebalancing. Maintaining service threads also increases computing resource contention. Experiment results in Section 8.1 show that Teseo suffers the most performance degradation in workloads with hotspots due to contentions during structure modifications. GTX overcomes these through 1) A cooperative design with no service thread except a single commit manager. GTX's structure modification (edge-deltas block consolidation) and garbage collection are handled by worker threads cooperatively during runtime (Sections 6 and 7). 2) GTX's state-based block-protection protocol designates one worker thread to clean one memory block to reduce contention. Worker threads do not compete to lock memory blocks (Sections 6.2 and 6.3).

Sortledton [52] is a graph data structure optimized for the access patterns popular in graph analytics. It organizes vertices in a 2-level vector index and stores sorted adjacency lists in vectors and edge blocks for better read performance at the expense of maintaining the sort during updates. For hub vertices, Sortledton has a concurrent unrolled skiplist [81] to store blocks of edges in sorted sets, where each skiplist element is an edge block. Sortledton's transactions support scan and set operations on adjacency lists and concurrent vertex/edge updates. However, Sortledton requires transactions to know in advance their read and write sets, and lock them following a global order beforehand through vertex locking. Transactions need to complete all their reads before doing any writes. GTX's ACID transaction protocol does not require knowing read sets and write sets beforehand. A GTX transaction can execute reads and writes in any order during runtime (Section 6). Moreover, GTX's delta-chains-based concurrency control eliminates vertex-centric locking (Section 6.1) to provide better concurrency.

Latch-free indexes and systems [24, 25, 42, 66, 67, 70, 71, 74, 98, 107, 109] use atomic operations, e.g., *CAS* and *fetch_add*, to serialize concurrent operations. GTX uses atomic operations and delta-based multi-versions for concurrent lock-free reads and to resolve write-write conflicts. It has sequential memory blocks for adjacency lists as in [99, 111] to amortize the cost of delta allocations. Deltas are stored sequentially to reduce random access.

3 Overview of GTX

GTX manages and queries dynamic labeled property graphs [57], where edges can contain labels, e.g., as in [111]. Fuchs et al. [52] microbenchmark the popular access patterns in graph workloads. Major takeaways from their study are: (1) Storing neighbors of all vertices together (sequential vertex access) is beneficial but is not necessary. (2) Sequential access to each vertex's adjacency list enables better performance. (3) A dense vertex identifier domain has better performance for graph analytics than a sparse domain. (4) Some graph algorithms access vertices in random order and require a vertex index. Given these findings, we make several design decisions. GTX uses an adjacency-list-like structure, and stores edges of a vertex sequentially in memory blocks (Section 4.2). GTX uses a vertex index for fast lookup, and an atomic integer on a dense domain to manage vertex IDs. For atomicity, consistency, and isolation, graph updates and analytics are executed under read-write or read-only transactions with snapshot isolation [26, 106]. Transactions support vertex

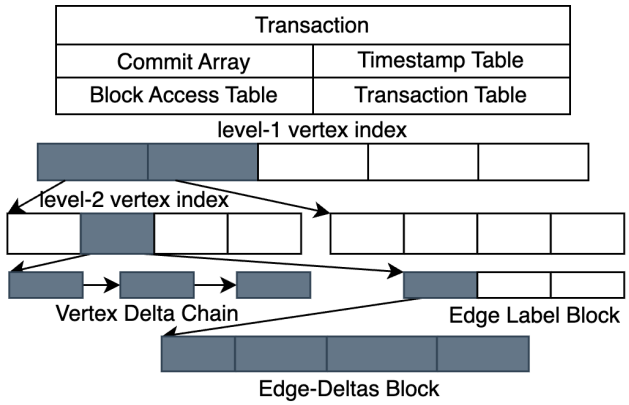


Fig. 1. GTX Overview: Transactions and graph store

and edge reads, adjacency list scans, and writes (insert, update, and delete). Graph analytics invoke adjacency list scans iteratively via a read-only transaction. GTX supports graphs with uniform and power-law edge distribution but is optimized for power-law graphs. Figure 1 gives an overview of GTX's storage and constructs for transactions. The colored portion in gray corresponds to GTX's graph storage and the uncolored portion corresponds to GTX's transaction protocol with auxiliary data structures.

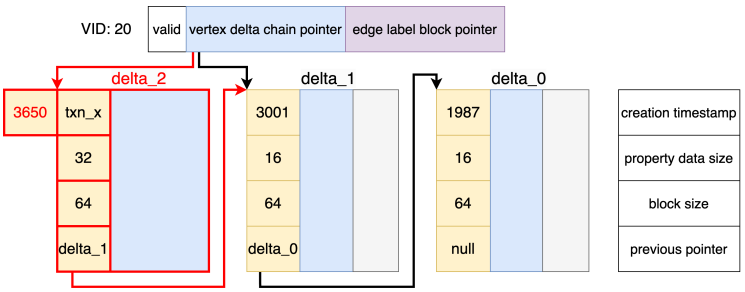


Fig. 2. Vertex Delta and Operations

4 GTX Storage

GTX is a main-memory MVCC graph data system. Similar to other *MVCC*-based systems [39, 42, 52, 67, 75, 111], GTX stores vertex and edge updates into deltas, links deltas of the same object, and manages deltas' lifespan and visibility through timestamps. GTX uses a multi-versioning design similar to Hekaton [42], Cicada [75], and LiveGraph [111]. Each of GTX delta's visibility and lifespan are managed by creation and invalidation timestamps. A delta's lifespan is the time between its creation and its invalidation timestamps (that is infinity for the latest version delta). A transaction T is assigned a read timestamp (rts), and can read all deltas whose lifespan overlaps T's rts. T creates deltas to update vertices and edges. Deltas of the same vertex or edge are linked forming version chains. The beginning of the version chain is the most recent delta. GTX enforces that every delta has the same creation timestamp as the invalidation timestamp of the delta right after it.

4.1 Vertex-centric Index

GTX uses a dense vertex domain, and maintains a vertex index. Figure 1's colored portion (in gray) illustrates GTX's storage. Sortledton shows the benefits of having vector-based vertex table for

graph storage [52]. GTX designs a similar latch-free two-level vector-based vertex index. Level-1 stores pointers to Level-2 in a fixed-size array. Level-2 has vertex index vector segments of the same size (or growing sizes). A transaction atomically *fetch_adds* a global offset variable, and uses the return value as the ID of a new vertex, say v. The corresponding Level-2 entry in the vertex index points to v and its edge versions. The initial vertex index has only 1 pointer in Level-1 and allocates 1 Level-2 vector. When the vector is out of entries, the transaction creating the first vertex that resides in the next vector allocates the next Level-2 vector, and claims its first entry. Other concurrent transactions creating vertices that need to be in the next vector will wait until the corresponding pointer in Level-1 is set. When GTX rarely runs out of space to allocate more vertex index entries, it allocates a new Level-1 array with double the size, and copies all vector pointers there similar to dynamic array expansion. The benefits are two-fold. Breaking the Level-2 vectors into segments managed by pointers reduces memory waste of allocating a single large vector. Level-2 vector extension avoids explicitly copying/touching vector contents and only requires appending a new segment. Meanwhile, GTX still maintains $O(1)$ vertex lookup. v's vertex index entry can be retrieved at $V[i][j]$: $i = v/c$ and $j = v \ mod \ c$, where c is the size of each Level-2 vector segment.

A vertex index entry stores an atomic pointer to the head of the vertex delta-chain. Figure 2 shows an example of Vertex 20 (V20), ignoring the parts in red for now. V20 has 2 versions, $delta\_0$ created and invalidated at timestamps 1987 and 3001 and $delta\_1$ created at timestamp 3001. Each vertex delta stores how many bytes its vertex property takes, and stores the property data inline. $delta\_0$ does not store its invalidation timestamp and infers it from $delta\_1$'s creation timestamp. Each vertex delta stores a pointer to the previous delta of the same vertex, forming a delta chain.

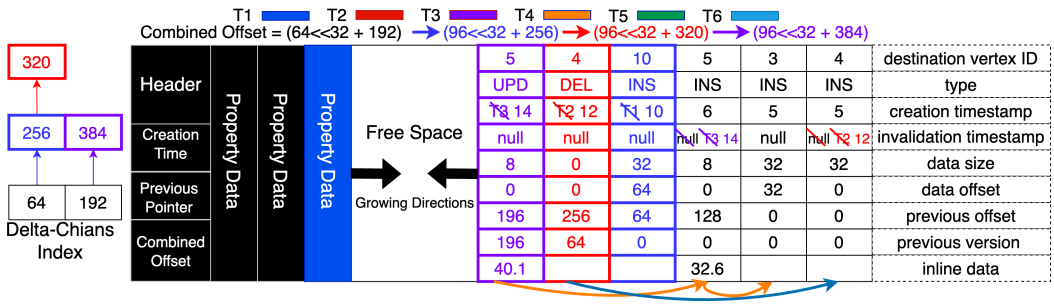


Fig. 3. Edge-deltas Block and Edge Operations

4.2 Edge Deltas Storage

The second pointer in each vertex index entry points to an edge label block. GTX stores all edges of the same source vertex and label in an edge-deltas block. Due to space limitation, we omit details of how to manage latch-free label blocks. It stores one entry per label pointing to an edge-deltas block and its associated delta-chains index, version number, and state variable (Detailed in Sections 5, 6). The experiment datasets and our latter discussion assume the graph contains only 1 label and omits showing labels for brevity.

Figure 3's contents in black illustrate V20's edge-deltas block and delta-chains index. GTX's edge-deltas block has similar storage layout to that of LiveGraph [111] and improves it with atomic memory allocation, edge delta-chains, and hybrid property data storage. At the lowest address, the header stores the block's metadata, e.g., creation time. Then, a property data storage section stores each edge delta's variable-sized property data. Fixed-sized edge deltas are stored at the opposite end of the block. Edge deltas are logically linked in different edge delta-chains. Each block has a delta-chains index whose i^{th} entry stores the offset of the i^{th} edge delta-chain's head. The

combined_offset combines the offsets of the latest edge delta and property data in a 64-bit integer that can be atomically updated by the hardware without a latch. It enables atomically allocating memory for both the edge delta and its property in a single operation. Its higher 32 bits store the offset of the property data region and the lower 32 bits store the offset of the edge deltas region, extending from single direction space preallocation in Open Bw-tree [99], and latch-based allocation in LiveGraph [111]. Silo [98] and Tictoc [107] also store multiple objects in a single atomic word to enable atomic multi-reads and multi-writes.

Each edge delta takes 1 cache line (64 bytes) to avoid false sharing among concurrent writers. It stores a write operation and the creation and invalidation timestamps for multi-versioning. An edge delta's *previous_offset* points to the previous edge delta on the delta chain. At an edge-deltas block's creation time, GTX determines its edge delta-chain number, where transactions append edge deltas to different delta chains using a hash function. Hekaton [42, 67] also maintains linked lists of all versions in the same hash bucket (may be different objects). GTX's novelty lies in the combination of pointer-based delta linking and sequential storage preallocation. Its delta linking serves as an index for efficient single edge delta lookup and all edge deltas of the same vertex are still stored sequentially in one memory block to preserve localities in adjacency list scans. Such a locality-preserving sequential storage proves to be beneficial for the graph analytics [39, 52, 76, 94, 111]. Finally, GTX delta chain storage and delta-chains index are tightly coupled with concurrency control (Sction 6.1). Each edge delta also stores the offset of the same edge's previous edge delta to support efficient visible version lookup. Given a target edge e, a reader transaction locates e's edge delta-chain, and uses edge deltas' *previous_offset* and *previous_version_offset* to locate e's visible version. Edge delta's data size and data offset locate its edge property data. We extend Cicada's best-effort inlining [75] and take advantage of "wasted" space in each cache line-aligned edge delta to design a *hybrid* storage scheme for its variable-sized property. If the property size is small, e.g., $\leq$ 16 bytes, it is stored in the cache line-aligned edge delta to save space and reduce random memory access, else it is stored in the data region. In contrast to Cicada's inlining that only applies to one version and has management overheads, every GTX edge delta can store property data inline to save space and improve cache performance. Because each edge delta's inline data region is allocated regardless of the property size for cache-line alignment, there is no extra overhead for GTX's hybrid storage scheme.

In the example, V20 has 3 edges: $e(20, 4)$, $e(20, 3)$, and $e(20, 5)$ (*insert* edge deltas) and maintains 2 edge delta-chains. Its delta region takes 192 bytes and data region takes 64 bytes (*combined_offset*). $e(20, 4)$ and $e(20, 3)$ have 32 bytes of property data each stored in the data region (Offsets 0 and 32). $e(20, 5)$ stores its 8-byte weight inline in the edge delta. $delta\_chain\_0$ contains 1 edge delta $d(4, INS)$ at Offset 64, and $delta\_chain\_1$ contains 2 edge deltas $d(3, INS)$ and $d(5, INS)$ starting at Offset 192. The middle section is free space.

5 Graph Transaction Operations

GTX transactions support vertex and edge insert, delete, and update, single vertex/edge lookup, and adjacency list scan. In this section, we persent transaction operations, and defer transaction management (e.g., concurrency control, commit) to Section 6. Let cts, its, rts, and wts be the creation, invalidation, read, and write/commit timestamps, respectively. At creation time, each transaction gets an rts and can see all deltas with overlapping lifespans. For writes, transactions create deltas with transaction IDs as timestamps, and update them with wts at commit time.

5.1 Vertex Read and Update

GTX transactions update vertices through creating vertex deltas, and read them by finding/reading visible deltas using atomic load and *CAS*. Refer to the same example of V20 in Figure 2 including the

red parts. Assume that Transaction t with $rts = 2000$ looks up V20. After fetching V20's delta chain pointer from the vertex index, t checks the delta chain head $delta\_1$, and finds its cts is larger than t's rts. Thus, t uses the previous pointer to locate $delta\_0$ with $cts = 1987$. t determines $delta\_0$ is visible, and reads it. If a newer transaction $t2$ with $rts = 4000$ also reads V20, $t2$ will read $delta\_1$. Assume that a read-write transaction txn_x updates V20. Similarly, it loads the delta chain head as in the read operation. txn_x checks $delta\_1$'s cts, and makes sure it is not a transaction ID, nor larger than txn_x's rts. Else, txn_x immediately aborts due to a write-write conflict. txn_x creates a new vertex delta $delta\_2$ with the updated property, and stores txn_x as its cts. $delta\_2$ stores a pointer to $delta\_1$. Then, txn_x invokes *CAS* to update the vertex delta-chain pointer to point to $delta\_2$. If *CAS* fails, txn_x aborts due to write-write conflict. Else, txn_x commits at $wts = 3650$, and updates $delta\_2$'s cts to its wts. This applies to vertex insert and property update. Deletes are covered in Section 5.5.

While using a pointer-based delta chain and *CAS* is less efficient due to pointer chasing, random memory access and competing *CAS* [99], vertex updates are less frequent than edge updates [111], hence less conflicts from concurrent updates. Also, most transactions need access to only the latest version of a vertex, and will access it with a single pointer chase. Thus, the negative effects of this design are minimal. On the other hand, detecting write-write conflicts to the same vertex becomes simple. A failed *CAS* indicates a concurrent transaction has updated the vertex delta chain.

5.2 Edge Update

GTX transactions update edges by creating edge deltas (insert, update, and delete) in edge-deltas blocks. GTX supports directed graphs using 2 labels and 2 edge-deltas blocks per vertex for fanin and fanout edges. Undirected graphs store each edge twice in both vertices' edge-deltas blocks as in [39, 52].

Refer to the example of V20's edge-deltas block in Figure 3. Colors refer to transactions' updates, e.g., Transaction $T1$'s updates are in dark blue. Assume that $T1$ with $rts = 9$ updates $e(20, 10)$ with a new 32-bytes property. First, it calculates that $e(20, 10)$ belongs to $delta\_chain\_0$ by $10 \ mod \ 2 = 0$, where the number of delta chains in this block is 2. $T1$ reads the delta-chains index, finds the offset 64, and checks the edge delta at this location. $T1$ sees $d(4, INS)$ has a smaller cts than its rts, so it decides to lock $delta\_chain\_0$ (covered in Section 6.1). After $T1$ successfully locks $delta\_chain\_0$, it uses each edge delta's *previous_offset* to look for the previous version of $e(20, 10)$, that in this example does not exist. A lookup of the previous version is necessary to avoid creating duplicate edges and link versions together. Delta-chains index enables efficient lookup of edges. Since the new edge version has a 32-bytes property, $T1$ needs to store the property in the data region. Then, it invokes the delta allocation protocol (32 bytes for data and 64 bytes for delta regions) to update the *combined_offset*, and creates the corresponding edge delta with $cts = T1$, $type = INS$, *previous_offset* pointing to $d(4, INS)$, no *previous_version_offset*, data size equaling 32, and data offset equaling 64. When $T1$ is committing, it updates the delta-chains index entry to point to $d(10, INS)$, releases the delta-chain lock, gets its $wts = 10$, and updates $d(10, INS)$'s $cts = 10$.

The example also shows 2 other transactions $T2$ and $T3$ deleting $e(20, 4)$ and updating $e(20, 5)$ with a weight 40.1. $T2$ follows similar procedures as $T1$ with a few differences. $T2$ deletes an existing edge, so it fetches $d(4, INS)$ and puts transaction ID $T2$ in $d(4, INS)$'s its. $T2$ also writes $d(4, INS)$'s offset 64 in $d(4, DEL)$'s *previous_version_offset* to link versions. During commit, $t2$ writes its $wts = 12$ to $d(4, DEL)$'s cts and $d(4, INS)$'s its. $T3$ **updates** $e(20, 5)$ with a new 8-bytes weight and works similarly as $T2$. The difference is that the new weight is stored inline in $d(5, UPD)$. $T1$ stores it in the data region and $T2$'s $d(4, DEL)$ has no property data.

5.3 Edge Lookup

GTX supports single edge lookup given $(src, dst, label)$. The src and $label$ locate the edge-deltas block and its delta-chains index. Then, each reader transaction t calculates the edge delta-chain from dst, and reads its deltas. After finding a delta with a matching dst, t compares its rts with the delta's lifespan, and uses the *previous_version_offset* to locate the visible version if needed.

Refer to the same example in Figure 3. The colors of arrows represent the search path of each transaction. Transaction $T4$ with $rts = 15$ looks up $e(20, 3)$. After locating the edge-deltas block, $T4$ calculates $e(20, 3)$ belongs to $delta\_chain\_1$ by $3 \ mod \ 2 = 1$. $T4$ reads the offset (384) from the index, and finds $d(5, UPD)$, the head of $delta\_chain\_1$. Then, $T4$ reads $d(5, UPD)$'s *previous_offset* to search the delta chain. After reading $d(5, INS)$'s *previous_offset*, $T4$ finds $d(3, INS)$, and checks its cts and its. $T4$ determines $d(3, INS)$ is visible, and uses its data offset 32, and the data size 32 bytes to read the edge property accordingly. Transactions $T5$ with $rts = 15$ and $T6$ with $rts = 10$ both look up $e(20, 4)$. After reading $d(4, DEL)$ of $delta\_chain\_0$, $T5$ determines that $d(4, DEL)$ is visible, and reports $e(20, 4)$ being deleted. $T6$ sees $d(4, DEL)$'s $cts = 12$, larger than its $rts = 10$. Therefore, it uses $d(4, DEL)$'s *previous_version_offset* 64 to find $d(4, INS)$, verifies its rts overlaps with $d(4, INS)$'s lifespan, and reads the 32 bytes property data of $e(20, 4)$.

5.4 Adjacency List Scan

GTX supports sequential adjacency list scan given $(src, label)$. After locating the edge-deltas block, a transaction loads the *combined_offset* to determine where to start the scan, and scans every edge delta in the block. It compares its rts with each delta's lifespan, and returns visible edge deltas. We use an iterator model for adjacency list scans, and graph analytics directly accessing each block. In Figure 3, Transaction $T7$ with $rts = 20$ scans V20's adjacency list. It loads the *combined_offset*, determines the edge deltas start at Offset 384, and reads every edge delta from there. It reports V20 has edges $e(20, 5)$ with Weight 40.1, $e(20, 10)$ and $e(20, 3)$.

5.5 Vertex Delete

Vertex deletion is an expensive but necessary operation in graph systems. Recent graph database benchmarks LDBC-SNB-Interactive-v2 [14, 82] and LDBC-SNB-BI [13, 97] include deep cascading deletion operations. Deleting a vertex requires removing all its adjacent edges atomically. It preserves graph consistency by avoiding dangling edges. Here, we illustrate how to delete Vertex V20. To delete V20, Transaction td puts V20 in an exclusive deletion state, installs a special vertex-delete delta and scans V20's adjacency list. Then, td deletes these "reverse" edges in V20's neighbors' adjacency lists using edge deletion operations (Section 5.2). GTX currently only allows one vertex delete at a time, and we find this enough for vertex delete requirements, and it simplifies concurrency control. LDBC-SNB benchmarks have less than 1% writes as deletes, and vertex deletes are a minority among them [7, 82, 97]. More on vertex delete, including exclusive delete state, is presented in Section 6.6.

6 Transaction Management

GTX allocates worker threads to execute transactions. Each thread executes its own transactions while being cooperative, i.e., performs work on behalf of other transactions. For analytical workloads, GTX uses OpenMP threads [16] to collectively execute a single read-only transaction. GTX supports ACID transactions that can execute multiple read and write operations atomically and ensure no dangling or duplicate edges. GTX supports Snapshot Isolation (SI) [26] based on *MVCC* [106], the only supported isolation level in LiveGraph [111] and Teseo [39], and stronger than Neo4j's default

Read Committed [9]. We find SI sufficient for graph workloads. The graph benchmarks LDBC-SNB-Interactive-v1 [45] and v2 [82] require read committed and SI, respectively. GTX manages two variables for SI and MVCC: Global read (*gre*) and Global write (*gwe*) epochs as in LiveGraph [111]. Silo [98] has an epoch-based transaction protocol but GTX is only similar in resource management. A GTX transaction acquires its *rts* from *gre* when created, and gets assigned a *wts* from *gwe* at commit time. For SI, GTX transactions have no read-write conflicts, and only handles write-write conflicts. As in Section 5.1, we use *CAS* to detect vertex update conflicts. In Sections 6.1, 6.2, we discuss concurrency control for edge delta-chains and edge-deltas blocks. Every transaction can see all deltas created, but not invalidated by itself by storing and checking transaction IDs in delta timestamps. A GTX transaction t only updates the latest versions: t compares the latest delta's cts of the object t is to update. t can update only if the latest delta is not an in-progress delta of another transaction and the delta's lifespan overlaps $t.rts$. GTX supports redo logging and checkpoint for durability.

6.1 Adaptive Delta-Chain Locking

Section 5.2 shows that a transaction t needs to lock edge delta-chains before installing edge deltas. Locking delta chains enables concurrent updates to different edges in the same edge-deltas block and detect write-write conflicts: Updates to the same edges. Delta chain granularity locking is a middle ground between vertex and edge locks; providing better concurrency than vertex locks and reducing overheads by managing a lock per edge. GTX does not use a high-overhead lock manager but a locking bit in the delta-chains index offsets as in Silo and TicToc [98, 107]'s version timestamps. Each delta-chains index entry uses its most significant bit as the locking bit. To lock a delta chain, t checks if the locking bit is set, and uses a *CAS* to set the bit atomically. Using locking bits in index offsets enables updating both the locking bit and the delta chain head atomically during commit as in Silo [98] and TicToc [107] that release locks while installing new timestamps. If the bit has already been set by other transactions, or the *CAS* fails, t detects a write-write conflict and aborts. Section 6.3 shows delta chain locking's adaptivity to edge workload history.

6.2 Block Protection Protocol

GTX supports different access permissions on edge-deltas blocks using a block state-based protection protocol inspired by [74]. GTX manages a global data structure *block_access_table* with one entry per worker thread. When a worker thread accesses an edge-deltas block, it calculates the block ID through the block's source vertex ID and label, and stores it in its table entry. Each table entry is padded to take one cache line to avoid false sharing. GTX associates every edge-deltas block with an atomic state variable. GTX defines 5 states: *normal*, *overflow*, *consolidation*, *installation*, and *delete*. *normal* makes up the majority of an edge-deltas block lifetime where both reads and writes are allowed. All examples in Figure 3 assume blocks in State *normal*. States *overflow*, *consolidation*, and *installation* are for edge-deltas block consolidation (Section 6.3). *overflow* and *installation* provide mutual exclusion on an edge-deltas block, and *consolidation* makes the block read-only. *delete* is used for vertex delete, and prevents other transactions from accessing a block. The Block Protection protocol has 2 operations: *register_access* and *change_state*. When Transaction t accesses Edge-deltas Block b, t calls *register_access(b)* to examine b's state. If t's operation is compatible with b's state, t stores b's ID in t's *block_access_table* entry. Then, t rechecks b's state. If it has not changed, t can proceed. If b's state is incompatible with t's operation, t unregisters its block access by storing 0 in the table, and retries later. Upon finishing, t unregisters its block access. A worker thread th needing to change b's state updates the state to the new value. If the new state is exclusive, th scans the *block_access_table*. If th finds no threads accessing b, th executes the operations in the new state. Else, th continues scanning the *block_access_table* until b is "safe".

Similar functionalities can be achieved using shared-exclusive locks, but we argue that Block Protection Protocol is better-suited for GTX's graph workloads. GTX already handles edge and vertex-level write-write conflicts using locking bits and *CAS*. Most of the time, an edge-deltas block is in *normal* state, and a transaction t only needs to read the state variable, and to store the block ID in its *block_access_table* entry. t does not write to the shared memory variables to reduce cache invalidation across cores. Only during edge-deltas block consolidation and vertex deletion, t needs to modify shared state variables and scans the *block_access_table* but these are rare. We trade-off their extra costs for low overhead execution of the more common normal transaction operations.

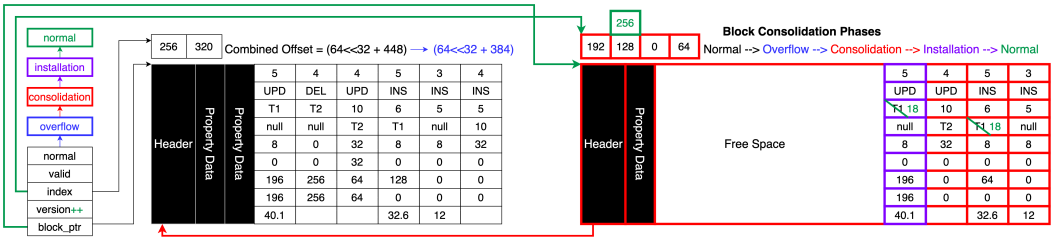


Fig. 4. Consolidation Example

6.3 Consolidation of an Edge-Deltas Block

GTX preallocates storage in edge-deltas blocks to reduce edge delta allocation overhead and provide sequential adjacency list access as in [99, 111]. As edge deltas build up, edge-deltas blocks become full. GTX needs to allocate new blocks to accommodate new edge deltas and recycle memory of the invalidated deltas. GTX edge-deltas block consolidation analyzes edges, and allocates a new block to store the latest version and future edge deltas. GTX consolidation has 2 features: Reduce interference with concurrent transactions via the Block Protection Protocol, and adjust parallelism in the adjacency list to accommodate workloads using adaptive delta-chain locking. Refer to the example in Figure 4. Operations in each phase are in different colors. V20's Edge-deltas Block b of Size 512 bytes is full with a 64 byte header, 6 edge deltas (384 bytes) and 64 bytes of property data with *combined_offset=(64«32+384)*. Two ongoing transactions $T1$ and $T2$ have edge deltas in b. Transaction t updates an edge and increments *combined_offset* to *(64«32+448)* to allocate a delta. $64 + 448 = 512$ is greater than the total space, so t starts b's consolidation. t changes b's state to *overflow*, waits till no other transactions are accessing b, and restores *combined_offset* to *(64«32+384)*. The corrupted overflow *combined_offset* cannot locate where edge deltas start. Resorting it in an exclusive state prevents concurrent transactions from accessing it. The atomic *fetch_add* ensures that only 1 transaction causes the overflow.

Analysis: t changes the state to *consolidation*, and analyzes b. Only edge updates are prohibited in this state. t scans b and finds latest version edge deltas: $d(4, UPD)$, $d(5, INS)$, and $d(3, INS)$. It also records $d(5, UPD)$ of $T1$ and $d(4, DEL)$ of $T2$ as in-progress. t calculates a new block size using the workload history: latest version and in progress edge deltas. The new block needs at least 418 bytes: 352 bytes to accommodate latest version and in progress deltas, and 64 bytes for the block header. Thus, t allocates a new block of size 1024 bytes; the smallest power of 2 that is at least 1.5x of 418, and 4 delta chains. GTX uses a heuristic to calculate the ratio, e.g., 1.5x, based on the delta timestamps for adaptivity (Details to be given in [110] due to space). Larger block sizes and more edge delta-chains can accommodate more edge deltas, and enable more concurrent updates. t allocates a new edge-deltas block b' and updates its header: pointing b''s previous pointer to b, and setting b''s creation timestamp to 10; the largest invalidation timestamp in the b. Transactions with

$rts < 10$ can access b. t also registers that b can be safely recycled after Timestamp 10. t writes the 3 latest version edge deltas to b', and updates the delta-chains index.

Installation: After installing all latest version edge deltas, t changes the block state to *installation* to synchronize transactions. GTX handles concurrency at delta chain level that may change after a consolidation. To avoid inconsistencies in delta chains after consolidation, *installation* synchronizes committing transactions before swapping the edge-deltas blocks. t updates delta-chains index entries to "install" its edge deltas during commit (Section 5.2). *installation* requires all transactions that have already "installed" their edge deltas in b to commit or abort before b continues consolidation, while committing transactions cannot install their edge deltas if b already has reached *installation*. t knows which delta chains and transactions to check based on the previous analysis. Assume that $T2$ has installed its edge delta but later aborts and $T1$ is still running. t waits till seeing $T2$'s final status(abort), and only writes $d(5, UPD)$ to b'. If $T2$ were to commit, t would also write $d(4, DEL)$ to b', and update the delta-chains index to point to $d(4, DEL)$. After installing all in-progress edge deltas, t increases the block version number, swaps *block_ptr* and index, and switches the block state back to *normal*. It allows other transactions, e.g., $T1$, to access the block. Later, $T1$ commits and updates its edge delta at $cts = 18$. Using the Block Protection Protocol ensures little overhead when the block is not overflowing, and improves concurrency during consolidation. Block consolidation adjusts block size and concurrency level to adapt to the workload: Hot(cold) blocks allocate more(less) memory and provide more(less) concurrency.

thread ID: 5	transaction ID	status	op_count	padding
counter: 18	5<<56 + 15	2031	2	
	5<<56 + 16	ABORT	0	
	5<<56 + 17	IN_PROGRESS	3	

Fig. 5. Thread 5's Transaction Table Partition

6.4 Lazy Update

GTX uses Lazy Update to reduce latency of transaction commits, and enforce SI [26, 106]. It distributes a transaction table to store transaction status: Each thread maintains its own transaction table, and can access other transaction tables based on thread IDs. We use the most significant 8 bits in transaction IDs to store worker thread information. Each transaction ID also has the most significant bit set to distinguish them from timestamps as discussed below. When a worker thread starts a new transaction, it calculates the transaction ID from its thread ID and a local counter, and stores the transaction entry in its transaction table. Each worker thread maintains every transaction t's *status* and *op_count* in its transaction table. *status* records t's current state: *In_Progress*, *Abort* that are predefined 64-bit integers or a positive commit timestamp. *op_count* records the number of deltas created by t. t has $status = In\_Progress$ and $op\_count = 0$ at creation time, and increments the *op_count* as it creates more deltas. If t aborts, its *status* is set to *Abort*. If t commits, its *status* stores its commit timestamp. Figure 5 gives an example of Thread 5's transaction table. Thread 5 has created 17 transactions and the table contains its $15\text{-}17^{th}$ transactions' information.

At runtime, a transaction t may access an edge-deltas block or vertex delta but encounters another transaction q's in-progress (edge) delta with $cts = q$. t examines q's status via q's ID, and updates the delta's *cts* (and previous version's *its*) to q's *wts* if q has committed. Then, t decrements q's *op_count* entry. Transaction table are circular arrays with reusable entries (if $op\_count = 0$). They allow $O(1)$ status lookup given transaction IDs. Similar approaches for managing delta commit, abort, and visibility by cooperative threads are used in CTR's fast recovery [22], Hekaton's cooperative resource management [42], and Hint Bits for reduced visibility check overheads [4, 87].

GTX uses Lazy Update for a new purpose, namely, high throughput and low latency transaction commit.

6.5 Hybrid Transaction Commit and Abort

GTX designs a high throughput low latency commit protocol for its read-write transactions inspired by Constant Time Recovery (CTR)'s fast recovery using a global table [22]. GTX's hybrid group-commit protocol combines Lazy Updates and group commits to support fast transaction commit while maintaining SI correctness [26, 106]. GTX has a commit manager thread that manages a global commit array, where each worker thread has an entry to register its committing transaction. A committing transaction t starts with a preparation phase: t makes its edge deltas the heads of their corresponding edge delta-chains, and unlocks them by updating the associated delta-chains index entries (Sections 5.2, 6.1). Section 6.3 introduces consolidation that may modify delta chains storage. Thus, during the preparation phase, t must register block access using the Block Protection Protocol, and checks block version. t is only allowed to install its edge deltas in *normal* and *consolidation* states. For other states, t has to wait and retry. If the block version has changed since t wrote its last delta, t has to "relock" all delta chains its edge deltas belong to. Only after locking all needed delta chains, t can install its edge deltas. Otherwise, t aborts. t executes commit preparation in a vertex ID order to avoid deadlocks. After t's preparation phase, future transactions can lock these delta chains, and lookup t's edge deltas using the delta-chains index. These operations are not needed for vertex deltas. Then, t puts a pointer to its transaction table entry in its commit array entry. The commit manager adds 1 to gwe as the group commit timestamp (wts), and scans the commit array. For each transaction, say t, in the array, the commit manager writes wts as t's status in the transaction table entry. After scanning the whole array, the commit manager *fetch_adds* gre by 1, **atomically committing** all transactions in this commit group at wts. After being assigned a wts, t eagerly updates its deltas' timestamps while the concurrently executing transactions can Lazy Update them, making up the hybrid commit protocol. Correctness-wise, transactions starting after the incremented gre are guaranteed to see deltas of the commit group. Performance-wise, it allows a group of transactions to commit without updating all their deltas, and the new gre to start immediately. It reduces group commit latency and allows the commit manager to work on the next commit group without waiting for the previous group to commit all its deltas. It reduces potential stalls of the new commit group caused by transactions with large write sets, and allows concurrent transactions to cooperate and parallelize these transactions' delta commits. It also decreases the synchronization between the commit manager and the commit group.

Transaction Abort: An aborting read-write transaction t writes *Abort* to its transaction table entry. Then, t iterates over its modified edge-deltas blocks, aborts all of t's deltas by writing *Abort* to their creation timestamps, and unsets any lock bits in the delta chains. t aborts all its vertex deltas by removing them from the delta chains.

6.6 State-aware Transaction Operations

Edge Operations. Transactions use the Block Protection Protocol for edge operations: states *normal* and *consolidation* for reads, and *normal* for writes. To accommodate block version change after consolidation, readers compare their rts with blocks' creation timestamp, and may need to use previous pointer to read an earlier edge-deltas block. Writers relock delta chains as in the commit preparation phase (Section 6.5) if they have edge deltas in the block with changed version and will abort if the relocking fails.

Vertex Delete. Block Protection Protocol has a an exclusive state *delete* for deep cascading vertex deletes at high priority. For example, when deleting V20 (Section 5.5), td sets each of V20's edge-deltas block's state to *delete* to delete the whole adjacency list. Concurrent transactions will **abort**

if they see *delete* on an edge-deltas block. After deleting all V20's edges in neighbors' edge-deltas blocks, *td* follows the same hybrid commit protocol. One difference is that *td* never aborts, and waits to lock delta chains to delete reverse edges of V20. GTX assigns vertex delete highest priority to satisfy privacy laws like European Union's General Data Protection Regulation that requires timely deletion of user data [82, 91, 97].

6.7 Transaction Durability

GTX supports transaction durability through redo logs and checkpoints. GTX implements group logging as in LiveGraph [111], and cooperative, parallel, and incremental delta-based fuzzy checkpointing inspired by Hekaton [42] and Silor [108]. Logging and checkpointing incur up to 22% performance overhead. We design and implement parallel recovery to reconstruct the graph after crashes. Due to space, we omit the design details of these 3 protocols and refers readers to our technical report [110].

6.8 Serializable Transactions

GTX optionally supports serializable read-write transactions using read-set validation [42, 75, 98]. While SI GTX only requires transactions to read a consistent snapshot, Serializable GTX requires a transaction's reads and writes to logically happen at the same time (its read-set stays unmodified during transaction execution) [42]. Each read-write transaction, say t, keeps a read-set R recording all vertices, edges, and adjacency lists t reads. When t commits, after the preparation phase to install edge deltas (Section 6.5), t validates R. For each $v, e \in R$, t checks if the vertex/edge delta chain stays unchanged by comparing the delta chain head's create timestamp. t validates edge-deltas blocks (adjacency lists) by checking whether they are updated with new deltas during t's lifetime. After t validates R, t commits as in Section 6.5. Else, t aborts. To accommodate the validation phase, the commit manager tracks the current commit group and enforces a serialization point between groups. It periodically enters a quiescent state and waits for all transactions in the current commit group to validate, then commit or abort. It starts a new commit group by incrementing global read/write epochs. Future transactions must wait to commit after the quiescent state. Read-only transactions are ordered by their *rts* to maintain serializability. Due to space, we cover only the major differences in transaction commit here, and present further details and optimizations for serializability support (e.g., early transaction abort during runtime) in our tech report [110].

7 Resource Management

Memory Allocation. GTX adopts the hybrid memory allocator from LiveGraph [111] to balance concurrent memory allocation and space utilization. It allocates a large chunk of memory at system start via *mmap*, and divides memory into blocks of varying powers-of-2 sizes (as in Buddy systems [63]). Each worker thread locally manages a pool of memory blocks smaller than a threshold and can allocate memory blocks from this pool without synchronization. Larger memory blocks are managed in a global pool. Due to the power-law nature of real-world graphs, most vertices are not hub vertices, i.e., do not have many edges. Thus, most of the vertex delta and edge-deltas block can be directly allocated from threads' private pools. Even when threads need to allocate a larger block, they unlikely request it at the same time. Thus, GTX's memory allocation protocol is mostly contention-free.

Cooperative Garbage Collection. *MVCC* creates garbage: versions that no current and future transactions can see. State-of-the-art *MVCC*/snapshot systems support garbage collections (GC) [22, 39, 42, 52, 67, 75, 94, 98, 111]. Cooperative garbage collection is used extensively in both relational and graph systems due to its good performance in coupling with normal transaction operations [42, 52, 67, 75, 98, 111]. It saves the overhead of maintaining additional garbage collection threads, and

provides timely recycling of garbage. GTX maintains a timestamp table (one entry per worker thread) to store the timestamps of running transactions. Worker threads register memory blocks as garbage during runtime, and periodically recycle the "safe" blocks. When a transaction commits or Lazy Updates a vertex delta, it **invalidates** the vertex's previous version. A worker thread invalidates the (old) edge-deltas block after consolidating it. Each worker thread, say w, maintains a local priority queue to store the invalidated vertex delta or edge-deltas blocks with their invalidation times. Periodically, w scans the timestamp table to find the "safe" (minimum) timestamp. w continuously recycles memory blocks from its garbage queue until the queue is empty or the current queue head's timestamp becomes too large. If a recycled block b is small, w stores b in its local pool, else b is sent back to the global pool [111]. This ensures that small blocks are allocated contention-free while large blocks return to the global pool for reuse by all threads. Silo [98], Hekaton [42], and Cicada [75] also register garbage, and perform cooperative safe timestamps-based garbage collection. GTX's GC is unique in its low overhead design. GTX recycles each outdated edge-deltas block as a whole instead of each invalidated edge delta individually, reducing garbage registration and recycling overhead. Each garbage registration is included in the normal operations: commit, Lazy Update or block consolidation. Thus, worker threads do not search for garbage. GTX worker threads have little interference when recycling garbage blocks. Each garbage priority queue is managed locally, and the timestamp table is cache-friendly and latch-free. GTX GC ensures that the safe memory block is inaccessible by other threads. Thus, there is no need for thread synchronization during GC. In contrast, state-of-the-art GCs generally execute at per-object granularity [75, 98], or interact with other threads through locks or atomic primitives [42, 75].

Systems	**Atomicity**		**Consistency**		**Isolation**		**Durability**
	single operation	multiple operations	no dangling edge	no duplicate edge	SI	Serializable	log+checkpoint
GTX	✓	✓	✓	✓	✓	✓	✓
LiveGraph	✓	✓	✓	✓	✓	✗	✓
Teseo	✓	✓	✓	✓	✓	✗	✗
Sortledton	✓	✓	✓	✓	✗	✓	✗
Spruce	✓	✗	✗	✓	✗	✗	✗

Table 1. Transactional Graph System Comparison

Graph Dataset	**|V|**	**|E|**	**Avg. Degree**	**Max Degree**
dota-league	61K	51M	1663.24	17004
graph500-24	9M	260M	58.7	406416
uniform-24	9M	260M	58.7	103
yahoo-songs	1.6M	256M	513.04	468425
edit-wiki	50M	572M	22.561	5576228
graph500-26	33M	1B	64.13	1003338
twitter	41M	1.46B	70.506	3081112

Table 2. Dataset Statistics

8 Evaluation

This study focuses on supporting high throughput read-write transactions with concurrent graph analytics.

Experiment Setup. Experiments run on a dual-socket machine with Intel(R) Xeon(R) Platinum 8368 @ 2.40GHz processors. It has 2 NUMA nodes and each NUMA node's CPU has 38 physical cores supporting up to 76 threads. We compile all systems with GCC 13.3.0 and O3 optimization flag,

and evaluate them over a single NUMA node with best effort, i.e., the evaluated system allocates memory locally, and uses remote memory only if its memory exceeds a single NUMA node. We conduct experiments with up to 64 worker threads. We choose to allocate fewer worker threads than one NUMA node's available threads to avoid causing computing resource contentions and induced inaccuracies and allow systems, e.g., Teseo [39] to spawn service threads. One exception is for insert experiments of *graph500-26* [12] and *twitter* [65] with over 1 billion edges. This machine's memory is not enough to evaluate all the systems. Thus, for these datasets, we evaluate all systems in a setup with Intel(R) Xeon(R) Platinum 8168 CPU @ 2.70GHz processors that has 12 cores supporting up to 24 threads and 387GB DRAM per NUMA node. To minimize NUMA effects, we run the experiments in a single NUMA node with up to 20 worker threads to avoid computing resource contention. Statistics of all graph datasets used in the experiments are given in Table 2. We use the same evaluation program based on the *LDBC* Graphalytics benchmark [20, 59] used in Teseo [39], Sortledton [52], and Spruce [94], and add more experiments. We evaluate against the transactional graph systems: LiveGraph [111], Teseo [39], and Sortledton [52]. Our evaluation focuses on SI isolation level, and we mainly evaluate the SI version of GTX. Note that LiveGraph and Teseo only support SI. In Section 8.4, we use an additional experiment to compare the SI and Serializable versions of GTX. Spruce [19, 94] lacks enough transactional support for the experiments. Its transactions can only update a single directed edge, or read a single vertex's neighborhood. To insert an undirected edge, Spruce creates 2 transactions (with two timestamps), breaking reciprocal consistency [35, 101]. Also, it creates 1 transaction per adjacency list scan, thus, failing to obtain a transaction consistent view of the whole graph. Thus, we exclude this system from the experiments. Table 1 compares these systems. Note that Sortledton requires transactions to know their read and write sets beforehand [52]. Experiments include a few instances of uniform graphs but the main focus is on power-law graphs as many real-world graphs exhibit such patterns [33, 34, 46, 95]. Experiment results follow the color codes in Figure 6. We have open-sourced GTX[1] and all experiments[2].

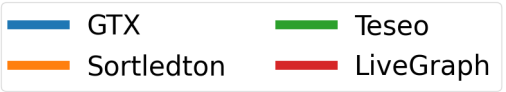


Fig. 6. *Bars Color Code in Experimental Results.*

8.1 Insert Performance

This experiment includes inserting uniform and power-law real-world and synthetic graphs with edges shuffled or according to timestamp order. Each system inserts an undirected edge by inserting 2 directed edges in a transaction and executes "checked" operations so that every edge insert involves a read to check if the edge exists. Thus, each transaction internally executes 2 edge lookups and 2 edge writes atomically. Thus, having efficient edge reads can improve transaction throughput.
Random-order Inserts. Evaluated systems insert edges from uniform and power-law real-world and synthetic graphs: *graph500-24, 26, uniform-24, dota-league* [12, 68] and *twitter* [65]. *graph500-26* and *twitter* have over 1 billion edges. Thus, we evaluate them on the machine with larger DRAM up to 20 worker threads. We have evaluated other graph datasets but their results are similar to the ones presented, and are omitted for brevity. The results are in Figure 7. GTX has over 10x better throughput than LiveGraph and 1.34x to 2x better throughput than the second-best Sortledton.

[1]https://github.com/Jiboxiake/GTX-SIGMOD2025
[2]https://github.com/Jiboxiake/gfe_driver_sigmod2025

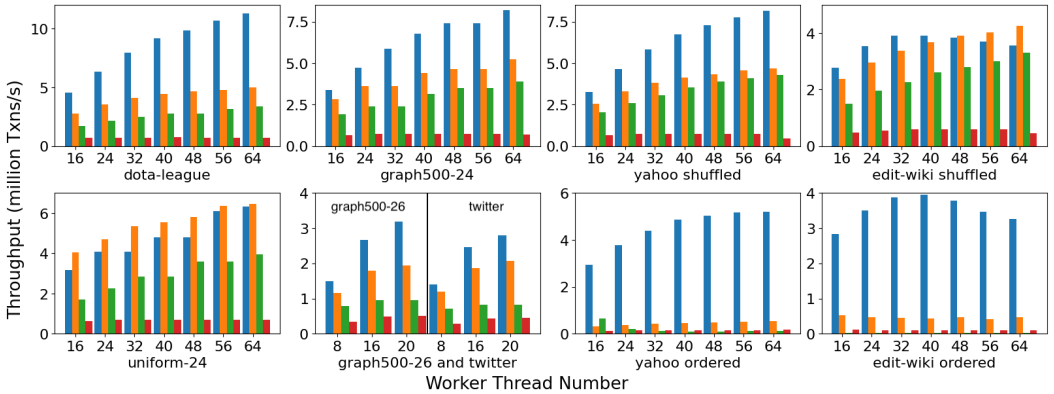


Fig. 7. Insertion Throughput

Timestamp-ordered Inserts. We insert edges of real-world graphs *yahoo-songs* and *edit-wiki* [65] based on their creation timestamps. Real-life power-law graphs not only have hub vertices (dataset *edit-wiki* has vertices with millions of adjacent edges) but also hotspots. Transactional graph systems have significant performance degradation when inserting edges in timestamp order [52]. Thus, we compare performances in both random and timestamp order insertions. Refer to Figures 7's *yahoo* and *edit-wiki*. As baseline, we evaluate inserting edges in random order. For *yahoo-songs*, GTX has the best scalability with up to 70% better throughput over Sortledton. For *edit-wiki*, GTX ranks the second and performs the best with medium parallelism degree. GTX does not scale well here at high worker thread numbers because its adaptive delta chains adjustment heuristic does not catch up well with large neighborhoods. GTX could have used a more aggressive heuristic to allocate larger edge-deltas block and increase delta chain numbers during consolidation, so more concurrent writers would be allowed per edge-deltas block. However, this will increase the size of allocated blocks and memory used by indexes, and may incur wasted spaces for low-degree vertices. We plan to explore more sophisticated heuristics in future works. In contrast, when inserting in the timestamp order, GTX performs the best over all competitor systems. The second-best Sortledton's throughput suffers up to 90% degradation. In contrast, GTX's throughput only drops by up to 30%. In *yahoo-songs*, GTX has up to 11x better throughput than Sortledton and in *edit-wiki*, GTX has up to 9x better throughput.

Performance Analysis. Experiment results show that GTX has the highest throughput in power-law graph updates and only GTX can maintain high performance under edge update hotspot and temporal localities. We attribute GTX's performance to its non-blocking latch-free graph store and low-overhead and adaptive delta-chain locking. GTX's exclusive locks are embedded in the delta-chains index with minimal overhead. GTX adjusts the number of delta chains of each edge-deltas block as edge updates arrive, and enables concurrent writes to the same vertex. Thus GTX achieves the highest performance in timestamp-ordered workload. For uniform graphs, Sortledton outperforms GTX for low parallelism degrees, but GTX's performance catches up as the number of concurrent worker threads increases. Since the graph is uniform, each vertex has about the same number of edges. Concurrent transactions are **not** likely updating the same adjacency lists. Thus, GTX's delta-chain locking becomes under-utilized, and the drawbacks of vertex-centric locking are mitigated because transactions are less likely to conflict in this case. However, for power-law graphs, Sortledton's threads block on concurrent updates on hub vertices and during a hotspot causing performance degradation. Ideally, Teseo should reduce the conflict in concurrently updating hub vertices by storing large neighborhoods in multiple segments. However, its sparse array segment

rebalance exclusively locks multiple adjacent segments. Thus, concurrent rebalance threads may conflict when locking the same adjacent segment or with other concurrent transactions. Teseo's segment locking may also incur false positives by locking vertices in the same segment. Thus, its throughput in timestamp-ordered experiment degrades by up to 99%. LiveGraph's performance suffers due to vertex-centric locking and the absence of edge indexes. For each edge insert, a LiveGraph transaction scans the edge block to check if the edge exists which is costly.
Lessons learned: Using latch-free techniques and low-overhead locking bits, having edge indexes, and breaking down vertex-centric locking to adapt workload enable higher transaction throughput.

Fig. 8. Concurrent Updates and Analytics: Transaction Throughput

Fig. 9. Concurrent Updates and Analytics: Graph Analytics Latency

8.2 Concurrent Updates and Analytics

We study how systems support concurrent update transactions and graph analytics simultaneously. We adopt graph update experiments from Teseo [39] and Sortledton [52], but add additional workloads. We generate log files based on the edges of the power-law graph *graph500-24* using *graphlog* [37]. The source graph has around 260M edges with 2.6B edge update logs. The 1st 10% edge logs are edge inserts that build the original graph. The next 90% are edge inserts and deletes. If a vertex has high degree in the graph, it has proportionally more edge update logs. The experiment ensures that graph size stays close to the initial size. All systems execute checked operations: edge inserts (deletes) only happen after checking the edge existence. Each system uses one NUMA node. If memory is not enough, it uses a remote node. LiveGraph's memory use is overly large, and could only process 20% of the logs. Thus, its experiments are conducted using 20% of the logs. We measure transaction throughput when running only updates. The results are similar to the transaction throughput results of this experiment and are omitted for brevity (One exception is Teseo's throughput increase due to its in-place updates and is the second best behind GTX). The mixed-workload experiment works as follows: We allocate 60 thread resources, and control the read/write ratio by assigning the threads either as write or OpenMP [16] read threads, e.g., 30 write and 30 read threads form a 50% write workload. Write threads continuously run update transactions while read threads run graph analytics. We could not report the results of Teseo as it runs into deadlocks (the same problem is reported in Sortledton's evaluation [52]). Sortledton has segmentation fault errors for some workloads, but we are still able to produce results. For analytics, we evaluate 1-hop (1-HN) and 2-hop neighbors (2-HN) that find 1-hop and 1-2 hop neighbors of

a set of vertices, and breadth first search (BFS), PageRank (PR), and single source shortest paths (SSSP) from Graphalytics [20, 59]. Similar experiments are in [52, 94] but they only run BFS and PR for certain thread setups.

Normal Update Distribution. We generate edge update logs and place them randomly in the edge log stream. Write threads run transactions executing edge logs, and concurrent graph analytics start at 10% of the update workload (after the source graph is loaded). The results are in Figures 8, 9. Sortledton crashes for several workloads. We redo these workloads and record the results for the successful runs. For 1-HN, PR, and SSSP, when the workload is read-heavy, e.g., 30% write, GTX's concurrent analytics can take at most 2× longer time than the best performing competitor. GTX has slightly higher transaction throughput than the second place Sortledton under these workloads. As the workload becomes more write-heavy, GTX's graph analytics performance catches up, and has the highest transaction throughput. For write-heavy workloads, e.g., 90% write, GTX has 1.46× to 1.7× higher transaction throughput than the second best system Sortledton, and performs the best and second best in SSSP and PR, respectively. GTX also maintains competitive performance in 1-HN against the other systems. Sortledton wins in BFS with significantly lower latency under all workloads. GTX is 1.13× to 1.75× faster than Sortledton in 2-HN under all workloads while achieving better transaction throughput (1.05× to 1.51×). LiveGraph exhibits good performance for graph analytics (around 0.99× to 1.54× faster than GTX), but its transaction throughput is significantly lower than the other 2 systems (GTX has 5.2× to 11.9× higher transaction throughput than LiveGraph).

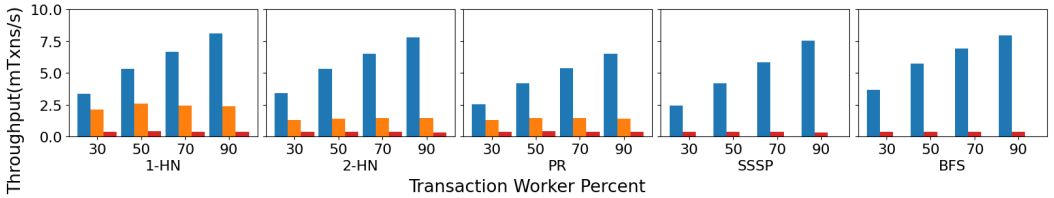


Fig. 10. Concurrent Update with Hotspot and Temporal Localities and Analytics: Transaction Throughput

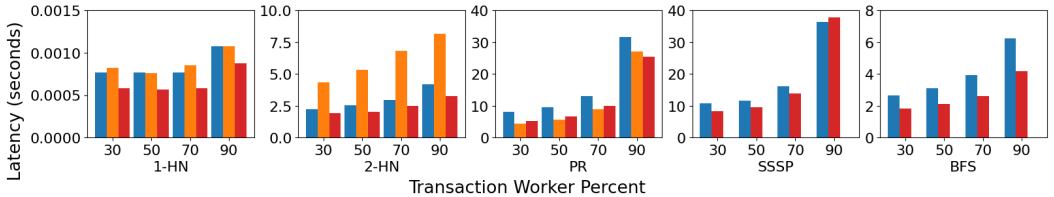


Fig. 11. Concurrent Update with Hotspot and Temporal Localities and Analytics: Graph Analytics Latency

Temporal Locality and Hotspot Update Distribution. We generate a set of edge logs with temporal localities and hotspots to mimic real-world scenarios. Transactions will more likely update the same vertex's adjacency list concurrently. This experiment has the same setting as the previous mixed-workload experiment except for having different edge logs. We study performance for concurrent reads and writes with temporal localities. Sortledton's segmentation fault persists for this hotspots workload. Thus, we only report available results in Figures 10, 11. All systems' graph analytics performances are not affected much by update hotspots. Sortledton and LiveGraph transactions degrade by up to 68.5% and 46% in throughput, respectively, for workloads with temporal localities and hotspots in update patterns. Only GTX maintains high transaction throughput (up to 5.3× and 22.3× better than Sortledton and LiveGraph, respectively) and is competitive in graph analytics.

Performance Analysis. GTX has the best transaction throughput while maintaining competitive graph analytics performance across different workloads. We attribute this to the following reasons. GTX supports delta-based *MVCC* that prevents read transactions from conflicting with write transactions. Thus, the interference between concurrent updates and graph analytics is mitigated. GTX's adaptive delta-chain locking adapts to workload temporal localities and hotspots to provide higher concurrency than vertex-centric locking systems. Thus, GTX can maintain high transaction throughput across all experiments. It stores and manages edge delta-chains of each adjacency list sequentially in a memory block. This preserves cache locality, reduces random memory access and cache misses, and facilitates prefetch. Previous works [39, 52, 111] show that sequential adjacency list storage is beneficial to adjacency list scan: the backbone for graph analytics. GTX implements graph algorithms (e.g., BFS, PR, and SSSP) directly accessing edge-deltas blocks, and is compiled within the system. It saves the cost of using adjacency list iterators and associated functional calls. Our experiments (not shown due to space) show that using an iterator is around 1.15× slower in graph analytics. Finally, GTX distributes the transaction table over worker threads to enable latch-free $O(1)$ access to transaction status for Lazy Updates, and the Block Access Table allows each reader and writer thread to register block-level access in its own cache-line-aligned entry without invalidating caches of other worker threads or modifying a shared lock.

Sortledton performs differently across experiments. It has the second best transaction throughput but it degrades for workloads with temporal localities and hotspots. For analytics, it performs the best in BFS. Sortledton uses exclusive lightweight latches for each adjacency list. BFS touches a smaller portion of the graph. Thus, update and read transactions are less likely to conflict, bypassing the negative locking effects. BFS requires each vertex's degree to initialize the algorithm. GTX and LiveGraph scan the vertex adjacency list to calculate this value. This dominates the BFS latency. Sortledton supports vertex degree versioning that allows transactions to get vertex degree efficiently, thus supporting efficient BFS. PR and SSSP require finding each vertex's degree, but they also need to scan vertices' adjacency lists multiple times. The algorithm's execution phase dominates the total latency, and thus Sortledton's advantages drop. As read threads scan the graph heavily, they become more likely to conflict with write transactions. 1-HN and 2-HN only requires adjacency list scans. In 2-HN, each vertex's 1-hop neighbors are not known a priori, causing Sortledton to have large locking overhead, and hence has worse performance than GTX. LiveGraph shows strong disparity between its transaction throughput and graph analytics. Its transaction throughput is several times worse than both GTX and Sortledton but its graph analytics perform better. LiveGraph uses a similar-styled *MVCC* as GTX's but only optimizes for sequential adjacency list scans [111]. It guarantees that each vertex's adjacency list is stored sequentially in memory and transaction reads and writes never conflict. Moreover, LiveGraph has smaller edge log entries compared to GTX's edge deltas. The compact sequential adjacency list storage enables good read performance. But LiveGraph suffers from its simple vertex-centric locking and lack of edge lookup indexes as discussed in Section 8.1 and has the worst transaction throughput.

Lessons learned: Limiting writes to shared memory variables, isolating updates in versions, and being cache-friendly reduce transaction overheads and interference. Having a sequential adjacency list and limiting function calls enable performant graph analytics.

8.3 Memory Consumption

GTX uses 64-byte edge deltas, so it uses more memory to load and store a graph. For power-law graph inserts, GTX uses around 2× to 3×, 0.9× to 2.1×, and 1.2× to 1.9× more memory than Sortledton, Teseo and LiveGraph respectively (Figures omitted for brevity). However, GTX's memory usage stays stable and consistent in more complex workloads with updates (invalidated versions), concurrent analytics, and hotspots. Figure 12 shows memory usage comparison of different systems

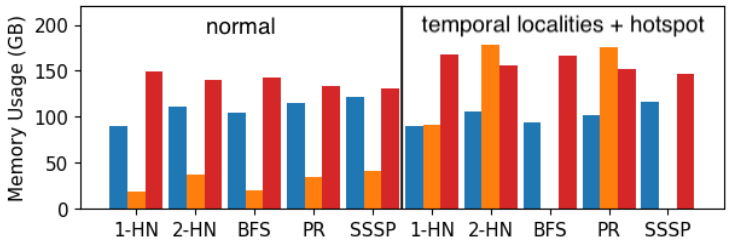


Fig. 12. Mixed-Workload Memory Consumption

under mixed-workload experiments[3]. It shows that GTX is efficient in memory allocation and garbage collection. LiveGraph uses more memory if the workload involves updates and analytics, while Sortledton's memory usage increases at most 5.17× in workloads with temporal localities and hotspots. GTX has the lowest memory consumption for these experiments.

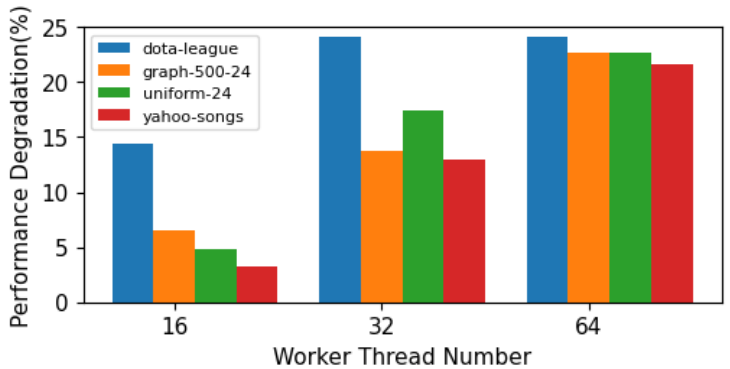


Fig. 13. Performance Degradation of Serializable GTX

8.4 Serializable vs. Snapshot Isolation GTX

We modify the insertion experiment (Section 8.1) to evaluate Serializable GTX's overhead over its Snapshot Isolation (SI) version using the same datasets and number of threads. Besides the checked edge insertion, each worker thread records the edge inserted in its last transaction. For each subsequent edge insert, the worker thread creates a new read-write transaction that first reads the previous edge inserted, and then inserts the checked edge. We report the average degradation using 4 of the datasets as in Figure 13. Serializable GTX has 3-25% throughput degradation vs. the SI version due to the following factors. Committing transactions and the commit manager need to synchronize in the quiescent state to enforce serialization between commit groups. Moreover, transactions need to recheck objects they read and will abort if they break serializability.

Lessons Learned: Serializable transaction support comes at a price of up to 25% performance degradation. LiveGraph [111] and Teseo [39] only support SI. Sortledton [52] supports serializability but requires known read and write sets and finishing all reads before writes. Because serializability support is costly and SI is sufficient for graph workloads (discussed in Section 6), serializable transactions should be used only when necessary.

[3]experiments conducted using 30 reader and 30 writer threads

9 Conclusions

GTX is a latch-free write-optimized transactional graph system that supports concurrent read-write transactions and graph analytics. It adopts an adjacency list format and a delta chain-based storage with delta-chains index. Its adjacency list design combines both the linked list-based delta store [70] and append-only delta updates [99, 111] to enable high update throughput, low-latency single edge lookup, and sequential adjacency list scan. GTX has high read and write concurrency using transactions, multi-versioning, and Lazy Update hybrid commit under snapshot isolation. Evaluation against state-of-the-art transactional graph systems shows that GTX has better performance for read-write transactions, and is several times better in handling real-world write workload with temporal localities. GTX performs well under mixed workloads when the workload is at least 50% writes, in which case, GTX has better read-write transaction throughput while its graph analytics latency remains competitive. For most of the write-heavy workloads, GTX significantly outperforms competitors in write throughput while maintaining close or even better graph analytics latency. Overall, GTX demonstrates that using latch-free techniques, avoiding vertex-centric locking, being cache-friendly, maintaining adjacency list-level edge indexes and sequential adjacency list storage, and being adaptive to the workload can enable high transaction throughput and better read-write concurrency in transactional graph systems.

Acknowledgments

Walid G. Aref acknowledges the support of the National Science Foundation under Grant Number IIS-1910216.

References

[1] [n. d.]. . https://graphbase.ai/
[2] [n. d.]. *RedisGraph*. https://redis.io/docs/latest/operate/oss_and_stack/stack-with-enterprise/deprecated-features/graph/
[3] 2013. *New Tweets per second record, and how!* https://blog.twitter.com/engineering/en_us/a/2013/new-tweets-per-second-record-and-how
[4] 2015. *Hint Bits*. https://wiki.postgresql.org/wiki/Hint_Bits
[5] 2019. *China's Singles' Day shopping spree sees robust sales*. http://www.xinhuanet.com/english/2019-11/11/c_138546429.htm
[6] 2020. *OrientDB*. https://orientdb.org/
[7] 2022. *LDBC benchmark data sets*. https://ldbcouncil.org/data-sets-surf-repository/snb-interactive-v2-updates.html
[8] 2023. *ByteDance*. https://www.bytedance.com/en/
[9] 2024. *Concurrent Data Access*. https://neo4j.com/docs/operations-manual/current/database-internals/concurrent-data-access/#transactions-isolation-lostupdates
[10] 2024. *Get Started with SAP HANA Graph*. https://developers.sap.com/group.hana-aa-graph-overview.html
[11] 2024. *JanusGraph*. https://janusgraph.org/
[12] 2024. *LDBC Graphalytics Benchmark (LDBC Graphalytics)*. https://ldbcouncil.org/benchmarks/graphalytics/
[13] 2024. *LDBC SNB Business Intelligence (BI) workload implementations*. https://github.com/ldbc/ldbc_snb_bi
[14] 2024. *LDBC SNB Interactive v2 workload implementations*. https://github.com/ldbc/ldbc_snb_interactive_v1_impls#ldbc-snb-interactive-v2-workload-implementations
[15] 2024. *Neo4j*. https://neo4j.com/
[16] 2024. *OpenMP*. https://www.openmp.org/
[17] 2024. *OQGRAPH Overview*. https://mariadb.com/kb/en/oqgraph-overview/
[18] 2024. *Oracle Big Data Spatial and Graph*. https://www.oracle.com/database/technologies/bigdata-spatialandgraph.html
[19] 2024. *Spruce*. https://github.com/Stardust-SJF/Spruce
[20] Wing Lung Ngai Stijn Heldens Arnau Prat-Pérez Thomas Manhardto Hassan Chafio Mihai Capotă Narayanan Sundaram Michael Anderson Ilie Gabriel Tănase Yinglong Xia Lifeng Nai Alexandru Iosup, Tim Hegeman and Peter Boncz. 2017. *LDBC Graphalytics Benchmark specification, v0.9.0.*

[21] Khaled Ammar. 2016. Techniques and Systems for Large Dynamic Graphs. In *Proceedings of the 2016 on SIGMOD'16 PhD Symposium* (San Francisco, California, USA) *(SIGMOD'16 PhD)*. Association for Computing Machinery, New York, NY, USA, 7–11. doi:10.1145/2926693.2929897

[22] Panagiotis Antonopoulos, Peter Byrne, Wayne Chen, Cristian Diaconu, Raghavendra Thallam Kodandaramaih, Hanuma Kodavalla, Prashanth Purnananda, Adrian-Leonard Radu, Chaitanya Sreenivas Ravella, and Girish Mittur Venkataramanappa. 2019. Constant time recovery in Azure SQL database. *Proc. VLDB Endow.* 12, 12 (aug 2019), 2143–2154. doi:10.14778/3352063.3352131

[23] Timothy G. Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. 2013. LinkBench: A Database Benchmark Based on the Facebook Social Graph *(SIGMOD '13)*. doi:10.1145/2463676.2465296

[24] Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. 2018. Bztree: A High-Performance Latch-Free Range Index for Non-Volatile Memory. *Proc. VLDB Endow.* 11, 5 (2018). doi:10.1145/3164135.3164147

[25] Greg Barnes. 1993. A Method for Implementing Lock-Free Shared-Data Structures. In *Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures* (Velen, Germany) *(SPAA '93)*. Association for Computing Machinery, New York, NY, USA, 261–270. doi:10.1145/165231.165265

[26] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A critique of ANSI SQL isolation levels. *SIGMOD Rec.* 24, 2 (may 1995), 1–10. doi:10.1145/568271.223785

[27] Maciej Besta, Marc Fischer, Vasiliki Kalavri, Michael Kapralov, and Torsten Hoefler. 2023. Practice of Streaming Processing of Dynamic Graphs: Concepts, Models, and Systems. *IEEE Trans. Parallel Distrib. Syst.* 34, 6 (June 2023), 1860–1876. doi:10.1109/TPDS.2021.3131677

[28] Maciej Besta, Robert Gerstenberger, Marc Fischer, Michał Podstawski, Jürgen Müller, Nils Blach, Berke Egeli, George Mitenkov, Wojciech Chlapek, Marek Michalewicz, and Torsten Hoefler. 2023. The Graph Database Interface: Scaling Online Transactional and Analytical Graph Workloads to Hundreds of Thousands of Cores. arXiv:2305.11162 [cs.DB]

[29] Maciej Besta, Robert Gerstenberger, Emanuel Peter, Marc Fischer, Michał Podstawski, Claude Barthels, Gustavo Alonso, and Torsten Hoefler. 2023. Demystifying Graph Databases: Analysis and Taxonomy of Data Organization, System Designs, and Graph Queries. *ACM Comput. Surv.* 56, 2, Article 31 (sep 2023), 40 pages. doi:10.1145/3604932

[30] Kai Zeng Bolin Ding and Wenyuan Yu. 2020. Alibaba Sponsor Talk at VLDB.

[31] Chiranjeeb Buragohain, Knut Magne Risvik, Paul Brett, Miguel Castro, Wonhee Cho, Joshua Cowhig, Nikolas Gloy, Karthik Kalyanaraman, Richendra Khanna, John Pao, Matthew Renzelmann, Alex Shamis, Timothy Tan, and Shuheng Zheng. 2020. A1: A Distributed In-Memory Graph Database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. 329–344. doi:10.1145/3318464.3386135

[32] Hongzhi Chen, Changji Li, Chenguang Zheng, Chenghuan Huang, Juncheng Fang, James Cheng, and Jian Zhang. 2022. G-Tran: A High Performance Distributed Graph Database with a Decentralized Architecture. *Proc. VLDB Endow.* 15, 11 (jul 2022), 2545–2558. doi:10.14778/3551793.3551813

[33] Rong Chen, Jiaxin Shi, Yanzhe Chen, Binyu Zang, Haibing Guan, and Haibo Chen. 2019. PowerLyra: Differentiated Graph Computation and Partitioning on Skewed Graphs. *ACM Trans. Parallel Comput.* (2019). doi:10.1145/3298989

[34] Audrey Cheng, Xiao Shi, Aaron Kabcenell, Shilpa Lawande, Hamza Qadeer, Jason Chan, Harrison Tin, Ryan Zhao, Peter Bailis, Mahesh Balakrishnan, Nathan Bronson, Natacha Crooks, and Ion Stoica. 2022. TAOBench: an end-to-end benchmark for social network workloads. *Proc. VLDB Endow.* 15, 9 (2022). doi:10.14778/3538598.3538616

[35] Audrey Cheng, Jack Waudby, Hugo Firth, Natacha Crooks, and Ion Stoica. [n. d.]. Mammoths Are Slow: The Overlooked Transactions of Graph Data. ([n. d.]).

[36] Miaomiao Cheng, Jiujian Chen, Cheng Zhao, Cheng Chen, Yongmin Hu, Xiaoliang Cong, Liang Qin, Hexiang Lin, Ronghua Li, Guoren Wang, Shuai Zhang, and Lei Zhang. 2023. ByteGAP: A Non-continuous Distributed Graph Computing System using Persistent Memory *(CEUR Workshop Proceedings)*. CEUR-WS.org. https://ceur-ws.org/Vol-3462/ADMS7.pdf

[37] Dean De Leo. [n. d.]. *graphlog*. https://github.com/whatsthecraic/graphlog

[38] Dean De Leo and Peter Boncz. 2019. Fast Concurrent Reads and Updates with PMAs *(GRADES-NDA'19)*. Association for Computing Machinery, New York, NY, USA, Article 8, 8 pages. doi:10.1145/3327964.3328497

[39] Dean De Leo and Peter Boncz. 2021. Teseo and the Analysis of Structural Dynamic Graphs. 14, 6 (feb 2021), 1053–1066. https://doi.org/10.14778/3447689.3447708

[40] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor Lee. 2019. TigerGraph: A Native MPP Graph Database. (01 2019).

[41] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. 2019. Low-Latency Graph Streaming Using Compressed Purely-Functional Trees. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. doi:10.1145/3314221.3314598

[42] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL server's memory-optimized OLTP engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 1243–1254. doi:10.1145/2463676.2463710

[43] Ayush Dubey, Greg D. Hill, Robert Escriva, and Emin Gün Sirer. 2016. Weaver: A High-Performance, Transactional Graph Database Based on Refinable Timestamps. *Proc. VLDB Endow.* 9, 11 (jul 2016), 852–863. doi:10.14778/2983200.2983202

[44] David Ediger, Rob McColl, Jason Riedy, and David A. Bader. 2012. STINGER: High performance data structure for streaming graphs. In *2012 IEEE Conference on High Performance Extreme Computing*. 1–5. doi:10.1109/HPEC.2012.6408680

[45] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. 2015. The LDBC Social Network Benchmark: Interactive Workload *(SIGMOD '15)*. doi:10.1145/2723372.2742786

[46] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. 1999. On Power-Law Relationships of the Internet Topology *(SIGCOMM '99)*. doi:10.1145/316188.316229

[47] Wenfei Fan, Tao He, Longbin Lai, Xue Li, Yong Li, Zhao Li, Zhengping Qian, Chao Tian, Lei Wang, Jingbo Xu, Youyang Yao, Qiang Yin, Wenyuan Yu, Jingren Zhou, Diwen Zhu, and Rong Zhu. 2021. GraphScope: a unified engine for big graph processing. 14, 12 (jul 2021), 2879–2892. doi:10.14778/3476311.3476369

[48] Zhuochen Fan, Yalun Cai, Zirui Liu, Jiarui Guo, Xin Fan, Tong Yang, and Bin Cui. 2024. CuckooGraph: A Scalable and Space-Time Efficient Data Structure for Large-Scale Dynamic Graphs. *CoRR* abs/2405.15193 (2024). https://doi.org/10.48550/arXiv.2405.15193

[49] Guanyu Feng, Zixuan Ma, Daixuan Li, Shengqi Chen, Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2021. RisGraph: A Real-Time Streaming System for Evolving Graphs to Support Sub-Millisecond Per-Update Analysis at Millions Ops/s *(SIGMOD '21)*. doi:10.1145/3448016.3457263

[50] Xiyang Feng, Guodong Jin, Ziyi Chen, Chang Liu, and Semih Salihoğlu. 2023. KÙZU Graph Database Management System. CIDR.

[51] Soukaina Firmli, Vasileios Trigonakis, Jean-Pierre Lozi, Iraklis Psaroudakis, Alexander Weld, Chiadmi Dalila, Sungpack Hong, and Hassan Chafi. 2020. CSR++: A Fast, Scalable, Update-Friendly Graph Data Structure. doi:10.4230/LIPIcs.OPODIS.2020.17

[52] Per Fuchs, Domagoj Margan, and Jana Giceva. 2022. Sortledton: A Universal, Transactional Graph Data Structure. *Proc. VLDB Endow.* 15, 6 (feb 2022), 1173–1186. doi:10.14778/3514061.3514065

[53] Sanchit Garg, Trinabh Gupta, Niklas Carlsson, and Anirban Mahanti. 2009. Evolution of an Online Social Aggregation Network: An Empirical Study. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement (IMC '09)*. doi:10.1145/1644893.1644931

[54] Pranjal Gupta, Amine Mhedhbi, and Semih Salihoglu. 2021. Columnar storage and list-based processing for graph database management systems. *Proc. VLDB Endow.* 14, 11 (jul 2021), 2491–2504. doi:10.14778/3476249.3476297

[55] Pankaj Gupta, Venu Satuluri, Ajeet Grewal, Siva Gurumurthy, Volodymyr Zhabiuk, Quannan Li, and Jimmy Lin. 2014. Real-Time Twitter Recommendation: Online Motif Detection in Large Dynamic Graphs. *Proc. VLDB Endow.* 7, 13 (aug 2014). doi:10.14778/2733004.2733010

[56] Mohamed S. Hassan, Tatiana Kuznetsova, Hyun Chai Jeong, Walid G. Aref, and Mohammad Sadoghi. 2018. Extending In-Memory Relational Database Engines with Native Graph Support. In *International Conference on Extending Database Technology*. https://api.semanticscholar.org/CorpusID:11389988

[57] Jim Webber Ian Robinson and Emil Eifrem. 2015. *Graph Databases: New Opportunities for Connected Data (2nd ed.)*. O'Reilly Media, Inc.

[58] Borislav Iordanov. 2010. HyperGraphDB: A Generalized Graph Database. In *Web-Age Information Management*, Heng Tao Shen, Jian Pei, M. Tamer Özsu, Lei Zou, Jiaheng Lu, Tok-Wang Ling, Ge Yu, Yi Zhuang, and Jie Shao (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 25–36.

[59] Alexandru Iosup, Tim Hegeman, Wing Lung Ngai, Stijn Heldens, Arnau Prat-Pérez, Thomas Manhardto, Hassan Chafio, Mihai Capotă, Narayanan Sundaram, Michael Anderson, Ilie Gabriel Tănase, Yinglong Xia, Lifeng Nai, and Peter Boncz. 2016. LDBC Graphalytics: A Benchmark for Large-Scale Graph Analysis on Parallel and Distributed Platforms. 9, 13 (2016), 12 pages. doi:10.14778/3007263.3007270

[60] Muhammad Attahir Jibril, Hani Al-Sayeh, Alexander Baumstark, and Kai-Uwe Sattler. 2023. Fast and Efficient Update Handling for Graph H2TAP. In *Proceedings 26th International Conference on Extending Database Technology, EDBT 2023, Ioannina, Greece, March 28-31, 2023*. OpenProceedings.org, 723–736. doi:10.48786/edbt.2023.60

[61] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedbhi, Jeremy Chen, and Semih Salihoglu. 2017. Graphflow: An Active Graph Database *(SIGMOD '17)*. Association for Computing Machinery. doi:10.1145/3035918.3056445

[62] Alexander D. Kent, Lorie M. Liebrock, and Joshua C. Neil. 2015. Authentication graphs: Analyzing user behavior within an enterprise network. *Computers & Security* 48 (2015), 150–166. doi:10.1016/j.cose.2014.09.001

[63] Kenneth C. Knowlton. 1965. A Fast Storage Allocator. *Commun. ACM* 8, 10 (oct 1965), 623–624. doi:10.1145/365628.365655

[64] Pradeep Kumar and H. Howie Huang. 2020. GraphOne: A Data Store for Real-Time Analytics on Evolving Graphs. *ACM Trans. Storage* 15, 4 (2020). doi:10.1145/3364180

[65] Jérôme Kunegis. [n. d.]. *The KONECT Project.* http://konect.cc/networks/
[66] Geof Langdale. [n. d.]. *Lock-Free Programming.* https://www.cs.cmu.edu/~410-s05/lectures/L31_LockFree.pdf
[67] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. 2011. High-performance concurrency control mechanisms for main-memory databases. *Proc. VLDB Endow.* 5, 4 (dec 2011), 298–309. doi:10.14778/2095686.2095689
[68] Dean De Leo. 2020. *GFE Driver - Input graphs.* https://zenodo.org/records/3966439
[69] Jure Leskovec and Rok Sosič. 2016. SNAP: A General-Purpose Network Analysis and Graph-Mining Library. *ACM Trans. Intell. Syst. Technol.* 8, 1 (2016). doi:10.1145/2898361
[70] Justin Levandoski, David Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for New Hardware Platforms. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)* (2013 ieee 29th international conference on data engineering (icde) ed.). IEEE. https://www.microsoft.com/en-us/research/publication/the-bw-tree-a-b-tree-for-new-hardware/
[71] Justin Levandoski, David Lomet, Sudipta Sengupta, Ryan Stutsman, and Rui Wang. 2015. High Performance Transactions in Deuteronomy. In *Conference on Innovative Data Systems Research (CIDR 2015).* https://www.microsoft.com/en-us/research/publication/high-performance-transactions-in-deuteronomy/
[72] Changji Li, Hongzhi Chen, Shuai Zhang, Yingqian Hu, Chao Chen, Zhenjie Zhang, Meng Li, Xiangchen Li, Dongqing Han, Xiaohui Chen, Xudong Wang, Huiming Zhu, Xuwei Fu, Tingwei Wu, Hongfei Tan, Hengtian Ding, Mengjin Liu, Kangcheng Wang, Ting Ye, Lei Li, Xin Li, Yu Wang, Chenguang Zheng, Hao Yang, and James Cheng. 2022. ByteGraph: A High-Performance Distributed Graph Database in ByteDance. *Proc. VLDB Endow.* 15, 12 (2022). doi:10.14778/3554821.3554824
[73] Hongfu Li. 2024. GastCoCo: Graph Storage and Coroutine-Based Prefetch Co-Design for Dynamic Graph Processing. arXiv:2312.14396 [cs.DB] https://arxiv.org/abs/2312.14396
[74] Tianyu Li, Badrish Chandramouli, and Samuel Madden. 2022. Performant Almost-Latch-Free Data Structures Using Epoch Protection. In *Data Management on New Hardware* (Philadelphia, PA, USA) *(DaMoN'22).* Association for Computing Machinery, New York, NY, USA, Article 1, 10 pages. doi:10.1145/3533737.3535091
[75] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. 2017. Cicada: Dependably Fast Multi-Core In-Memory Transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17).* Association for Computing Machinery, New York, NY, USA, 21–35. doi:10.1145/3035918.3064015
[76] Heng Lin, Zhiyong Wang, Shipeng Qi, Xiaowei Zhu, Chuntao Hong, Wenguang Chen, and Yingwei Luo. 2024. Building a High-Performance Graph Storage on Top of Tree-Structured Key-Value Stores. *Big Data Mining and Analytics* 7, 1 (2024), 156–170. doi:10.26599/BDMA.2023.9020015
[77] Memgraph Ltd. [n. d.]. *Memgraph.* https://memgraph.com/
[78] Peter Macko, Virendra J. Marathe, Daniel W. Margo, and Margo I. Seltzer. 2015. LLAMA: Efficient graph analytics using Large Multiversioned Arrays. In *2015 IEEE 31st International Conference on Data Engineering.* 363–374. doi:10.1109/ICDE.2015.7113298
[79] Norbert Martínez-Bazan, M. Ángel Águila Lorente, Victor Muntés-Mulero, David Dominguez-Sal, Sergio Gómez-Villamor, and Josep-L. Larriba-Pey. 2012. Efficient graph management based on bitmap indices. In *Proceedings of the 16th International Database Engineering & Applications Sysmposium* (Prague, Czech Republic) *(IDEAS '12).* Association for Computing Machinery, New York, NY, USA, 110–119. doi:10.1145/2351476.2351489
[80] Prashant Pandey, Brian Wheatman, Helen Xu, and Aydin Buluc. 2021. Terrace: A Hierarchical Graph Container for Skewed Dynamic Graphs *(SIGMOD '21).* doi:10.1145/3448016.3457313
[81] K. Platz, N. Mittal, and S. Venkatesan. 2019. Concurrent Unrolled Skiplist. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS).* doi:10.1109/ICDCS.2019.00157
[82] David Püroja, Jack Waudby, Peter Boncz, and Gábor Szárnyas. 2023. The LDBC Social Network Benchmark Interactive workload v2: A transactional graph query benchmark with deep delete operations. arXiv:2307.04820 [cs.DB]
[83] Hao Qi, Yiyang Wu, Ligang He, Yu Zhang, Kang Luo, Minzhi Cai, Hai Jin, Zhan Zhang, and Jin Zhao. 2024. LSGraph: A Locality-centric High-performance Streaming Graph Engine. In *Proceedings of the Nineteenth European Conference on Computer Systems* (Athens, Greece) *(EuroSys '24).* Association for Computing Machinery, New York, NY, USA, 33–49. doi:10.1145/3627703.3650076
[84] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. 2018. Real-Time Constrained Cycle Detection in Large Dynamic Graphs. 11, 12 (2018). doi:10.14778/3229863.3229874
[85] Christopher Rickett, Kristyn Maschhoff, and Sreenivas Rangan Sukumar. 2024. Optimizing the Cray Graph Engine for performant analytics on cluster, SuperDome Flex, Shasta systems and cloud deployment. *Concurrency and Computation: Practice and Experience* 36 (01 2024). doi:10.1002/cpe.7982
[86] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. 2017. The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing. *Proc. VLDB Endow.* 11, 4 (2017). doi:10.1145/3186728.3164139

[87] HANS-JÜRGEN SCHÖNIG. 2023. *Speeding up things with hint bits.* https://www.cybertec-postgresql.com/en/speeding-up-things-with-hint-bits/

[88] Bin Shao, Haixun Wang, and Yatao Li. 2013. Trinity: a distributed graph engine on a memory cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 505–516. doi:10.1145/2463676.2467799

[89] Aneesh Sharma, Jerry Jiang, Praveen Bommannavar, Brian Larson, and Jimmy Lin. 2016. GraphJet: Real-Time Content Recommendations at Twitter. *Proc. VLDB Endow.* 9, 13 (2016). doi:10.14778/3007263.3007267

[90] Aneesh Sharma, Jerry Jiang, Praveen Bommannavar, Brian Larson, and Jimmy Lin. 2016. GraphJet: Real-Time Content Recommendations at Twitter. *Proc. VLDB Endow.* 9, 13 (2016). doi:10.14778/3007263.3007267

[91] Supreeth Shastri, Vinay Banakar, Melissa Wasserman, Arun Kumar, and Vijay Chidambaram. 2020. Understanding and benchmarking the impact of GDPR on database systems. *Proc. VLDB Endow.* 13, 7 (mar 2020), 1064–1077. doi:10.14778/3384345.3384354

[92] Sijie Shen, Rong Chen, Haibo Chen, and Binyu Zang. 2021. Retrofitting High Availability Mechanism to Tame Hybrid Transaction/Analytical Processing. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 219–238. https://www.usenix.org/conference/osdi21/presentation/shen

[93] Sijie Shen, Zihang Yao, Lin Shi, Lei Wang, Longbin Lai, Qian Tao, Li Su, Rong Chen, Wenyuan Yu, Haibo Chen, Binyu Zang, and Jingren Zhou. 2023. Bridging the Gap between Relational OLTP and Graph-based OLAP. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association. https://www.usenix.org/conference/atc23/presentation/shen

[94] Jifan Shi, Biao Wang, and Yun Xu. 2024. Spruce: a Fast yet Space-saving Structure for Dynamic Graph Storage. *Proc. ACM Manag. Data* 2, 1, Article 27 (mar 2024), 26 pages. doi:10.1145/3639282

[95] Kimmo Soramäki, Morten L. Bech, Jeffrey Arnold, Robert J. Glass, and Walter E. Beyeler. 2007. The topology of interbank payment flows. *Physica A: Statistical Mechanics and its Applications* (2007). doi:10.1016/j.physa.2006.11.093

[96] Wen Sun, Achille Fokoue, Kavitha Srinivas, Anastasios Kementsietsidis, Gang Hu, and Guotong Xie. 2015. SQLGraph: An Efficient Relational-Based Property Graph Store. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1887–1901. doi:10.1145/2723372.2723732

[97] Gábor Szárnyas, Jack Waudby, Benjamin A. Steer, Dávid Szakállas, Altan Birler, Mingxi Wu, Yuchen Zhang, and Peter Boncz. 2022. The LDBC Social Network Benchmark: Business Intelligence Workload. *Proc. VLDB Endow.* 16, 4 (2022). doi:10.14778/3574245.3574270

[98] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 18–32. doi:10.1145/2517349.2522713

[99] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. 2018. Building a Bw-Tree Takes More Than Just Buzz Words *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 473–488. doi:10.1145/3183713.3196895

[100] Todd Warszawski and Peter Bailis. 2017. ACIDRain: Concurrency-Related Attacks on Database-Backed Web Applications. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 5–20. doi:10.1145/3035918.3064037

[101] Jack Waudby, Paul Ezhilchelvan, Jim Webber, and Isi Mitrani. 2020. Preserving reciprocal consistency in distributed graph databases. In *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC '20)*. Association for Computing Machinery. doi:10.1145/3380787.3393675

[102] Victor Junqiu Wei, Raymond Chi-Wing Wong, and Cheng Long. 2020. Architecture-Intact Oracle for Fastest Path and Time Queries on Dynamic Spatial Networks *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3318464.3389718

[103] Brian Wheatman and Randal Burns. 2021. Streaming Sparse Graphs using Efficient Dynamic Sets. In *2021 IEEE International Conference on Big Data (Big Data)*. 284–294. doi:10.1109/BigData52589.2021.9671836

[104] Brian Wheatman and Helen Xu. 2018. Packed Compressed Sparse Row: A Dynamic Graph Representation. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. 1–7. doi:10.1109/HPEC.2018.8547566

[105] Daniel ten Wolde, Gábor Szárnyas, and Peter Boncz. 2023. DuckPGQ: Bringing SQL/PGQ to DuckDB. *Proc. VLDB Endow.* 16, 12 (aug 2023), 4034–4037. doi:10.14778/3611540.3611614

[106] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. 2017. An Empirical Evaluation of In-Memory Multi-Version Concurrency Control. *Proc. VLDB Endow.* 10, 7 (2017). doi:10.14778/3067421.3067427

[107] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. 2016. TicToc: Time Traveling Optimistic Concurrency Control. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1629–1642. doi:10.1145/

2882903.2882935

[108] Wenting Zheng, Stephen Tu, Eddie Kohler, and Barbara Liskov. 2014. Fast Databases with Fast Durability and Recovery Through Multicore Parallelism. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 465–477. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/zheng_wenting

[109] K. Zhou, G. Tan, and W. Zhou. 2018. Quadboost: A Scalable Concurrent Quadtree. *IEEE Transactions on Parallel & Distributed Systems* 29, 03 (2018). doi:10.1109/TPDS.2017.2762298

[110] Libin Zhou, Lu Xing, Yeasir Rayhan, and Walid. G. Aref. 2025. GTX: A Write-Optimized Latch-free Graph Data System with Transactional Support – Extended Version. arXiv:2405.01418 [cs.DB] https://arxiv.org/abs/2405.01418

[111] Xiaowei Zhu, Guanyu Feng, Marco Serafini, Xiaosong Ma, Jiping Yu, Lei Xie, Ashraf Aboulnaga, and Wenguang Chen. 2020. LiveGraph: A Transactional Graph Storage System with Purely Sequential Adjacency List Scans. 13, 7 (mar 2020), 1020–1034. doi:10.14778/3384345.3384351

Received October 2024; revised January 2025; accepted February 2025