Fast and Efficient Update Handling for Graph H^{2}TAP

Muhammad Attahir Jibril
TU Ilmenau, Germany
muhammad-attahir.jibril@tu-ilmenau.de

Hani Al-Sayeh
TU Ilmenau, Germany
hani-bassam.al-sayeh@tu-ilmenau.de

Alexander Baumstark
TU Ilmenau, Germany
alexander.baumstark@tu-ilmenau.de

Kai-Uwe Sattler
TU Ilmenau, Germany
kus@tu-ilmenau.de

ABSTRACT

Offloading graph analytics to GPU yields significant performance speedups. In heterogeneous hybrid transactional/analytical graph processing (graph H^{2}TAP), where each graph workload type is executed on the most suitable processor, transactions are executed on a CPU-based main graph and analytics are executed on a GPU-optimized graph replica. The problem that arises, as a result, is that updates by transactions on the main graph have to be particularly handled with respect to the graph replica.

In this paper, we present a fast and efficient approach to this update handling problem, based on a *delta store* optimized for graphs. The delta store is a differential graph store that captures the transactional updates, which are later propagated to the graph replica so that analytical queries are executed on the most recently committed version of the graph in accordance with freshness requirements. Our approach ensures consistency between the main graph and the replica. Our evaluation shows the performance advantage of our approach over existing HTAP approaches.

1 INTRODUCTION

Various real-world applications are rapidly generating data that is more intuitive to model as a graph, where data entities are represented as nodes/vertices interconnected by relationships/edges [35, 61, 62]. Storing, managing and analyzing graphs is a recent and important data management problem, for which several architectures implementing different data models are being used [2, 36, 65, 83]. Numerous native graph databases have been developed particularly targeting graphs that are not only dynamic in that their structure changes over time with the insertion and deletion of nodes and relationships, but also have *labels* and rich *properties* associated with the nodes and relationships [3]. Several graph data models are being adopted for such graphs, with the most common among them being the Labeled Property Graph or *property graph* [4].

There are typically two broad types of graph workloads: transactional and analytical workloads. *Transactional workloads* handled by graph databases include inserting and deleting nodes and relationships, updating their properties, retrieving nodes with specific labels and/or property values, traversing the neighborhood of certain nodes, exploring a portion of the graph filtered by specific relationship labels and/or property values etc. [23]. These workloads are latency-critical and executed in such a way as to maximize throughput. *Analytical workloads*, on the other hand, include numerous graph algorithms such as Breadth First Search (BFS), Single Source Shortest Path (SSSP), PageRank (PR), Weakly Connected Components (WCC) etc. [38]. It could also include Business-Intelligence-like queries that heavily involve complex grouping and aggregation operations [71]. Analytical workloads are compute-intensive and long-running. The execution of an analytical task is thus parallelized to minimize latency. Although there exist several graph processing frameworks specialized only for *graph analytics* [36], however, some graph databases such as Neo4j [1] also support graph analytics in addition to transactional workloads. Furthermore, accelerating graph analytics with GPU delivers significant performance benefits for numerous graph algorithms [9, 10, 12, 13, 21, 22, 33, 41, 53, 54, 58, 68, 72, 75, 81, 82]. As a result, several GPU-based graph analytics frameworks have been developed, allowing to leverage GPUs for graph analytics [19, 20, 25, 27, 50, 59, 74, 77, 78].

There is a growing interest in graph systems that execute transactions and analytics concurrently on the same graph – termed hybrid transactional/analytical processing (HTAP) [28, 84]. These are fuelled by the increasing real-time requirements of various use cases [51, 66, 73, 82]. Systems for property graphs are already recently being developed for executing transactional and analytical workloads concurrently [84]. To realize HTAP graph databases, one option is to have storage structures optimized for property graphs (e.g. Neo4j's storage design of fixed-sized records [1]) and support concurrent execution of both transactional workloads and graph analytics on CPU. However, this forgoes potential performance speedup as GPU is not used to accelerate graph analytics in this case. Additionally, this option suffers from the limitations of single-store HTAP systems such as interference and lack of workload-specific optimizations (see Section 2.2 and Section 6.7 for more detail). Another option is a GPU-based HTAP graph database where both the analytics and the transactional workloads are executed on GPU completely [34]. This poses optimization issues for property graphs and transactional workloads because the data structure and algorithm optimizations for graph analytics on GPU target *structural graphs* (nodes interconnected by relationships with optional values like weights), as graph analytics target often only the graph *topology* and not labels and rich properties. It has been shown that in heterogeneous HTAP (H^{2}TAP), which is a system architecture aimed at emerging hardware where different processor types are exploited to match different workload types with their ideal processor types, transactional workloads are more suited to being executed on CPU while analytics are better executed on GPU [5, 56, 57]. Therefore, in as much as the high bandwidth and massive parallelization potential of GPU is to be leveraged by HTAP graph databases for accelerated analytics, the graph would inevitably be stored in two different representations. One is the main graph on CPU with a storage layout optimized for transactional updates on property graphs [39]. The other is the replica on GPU represented in a format optimized for graph analytics on structural graphs. Ultimately, this gives rise to *update propagation.* The accompanying challenges are:

(1) Propagating the transactional updates from the main graph to the replica with minimal overhead in order to preserve the overall system performance.
(2) Complying with HTAP freshness requirements in that the execution of analytics should be done on the most recently committed version of the graph.
(3) Guaranteeing data consistency between the main graph and the replica.

The graph replica on GPU would ideally be stored using a dynamic data structure for GPU-based graph analytics that are suitable for updates [7, 16, 31, 43, 64, 79, 80]. However, it could also be that the graph replica is in a static data structure, especially since the underlying data structures in the majority of GPU-based graph analytics frameworks [19, 20, 25, 27, 50, 59, 74, 77, 78] are static data structures such as the sparse matrix formats [10, 43, 64] – with the Compressed Sparse Row (CSR) being the most widely used among these static data structures [14, 30]. With regards to the dynamic data structure, to the best of our knowledge, there is no work on handling updates in a graph HTAP setting leveraging GPU, by way of propagating the updates from the main graph to the dynamic-data-structure-based graph replica on GPU. As for static data structures, they are not easily – if at all – updateable and, therefore, a complete rebuild of the data structures is resorted to, whenever any part of the graph gets updated [16, 31, 43, 64]. For example, on our GPU server (ref. Section 6.1), it takes about 3 seconds to execute the SSSP algorithm using the Gunrock [78] framework on a CSR generated from the Linked Data Benchmark Council's Social Network Benchmark (LDBC SNB) [3] graph data at *scale factor* 10. Assuming that the graph is updated, however, it takes 33 seconds to rebuild the CSR before executing the SSSP again on the new version of the graph, i.e. 11× more than the SSSP execution time. As the previously mentioned works have tried to optimize the performance of graph algorithms on GPU, the bottleneck now lies in propagating the updates, as this experiment shows. Rebuilding the CSR drastically degrades the overall system performance. And without this rebuild, data freshness is lost on the analytics.

In this paper, we address the earlier-mentioned three challenges of update propagation in the context of graph H^{2}TAP, where analytics are executed on a GPU-optimized replica of a transactionally updated main property graph. We propose a novel graph-based *delta* approach built upon a differential store or *delta store.* Our approach consists in capturing transactional updates to the main graph on CPU as deltas, which are stored in the delta store and propagated, later on, to update the graph replica on GPU while ensuring graph consistency. This way, the analytics are executed on the recently committed snapshot of the graph as per freshness requirements. Our approach is optimized for fast and efficient (in terms of delta store size) update storage and update propagation in graph H^{2}TAP for both the dynamic and static GPU-based graph data structures. We refer to the storage and propagation of updates combined as *update handling.* In summary, we make the following contributions:

- We propose a delta approach for update handling in graph H^{2}TAP (ref. Section 4.2).
- We present an efficient implementation of an append-only graph delta store for *storing* transactional graph updates[1]. Our delta store has a compact CSR-like layout that reduces memory overhead (ref. Section 5.1).
- We present a low-latency mechanism for *propagating* the updates from the delta store to the GPU (ref. Section 5.2).
- As part of the update propagation, we introduce a mechanism to enforce consistency between the main graph and the GPU-based graph replica (ref. Section 5.3).
- For cases where the replica is stored in a static data structure, we demonstrate a way to merge the updates from the delta store to the static data structure (ref. Section 5.4).

[1]https://dbgit.prakinf.tu-ilmenau.de/code/poseidon_core

2 BACKGROUND

2.1 Data Structures for GPU Graph Analytics

Graphs are widely represented using sparse matrices [29]. CSR is the most commonly used sparse-matrix format [30]. For an *adjacency matrix M* representing the topology of a graph, a CSR representation of the graph essentially stores the non-zero entries of M and their column indices in three one-dimensional arrays: the *edge values* array stores the actual non-zero entries, the *column indices* array stores the indices of the columns of those entries, and the *row offsets* stores the offsets of the values (in the first two arrays) for each row in M. These graph structures are static [14] and generally expensive to update - with the highly compressed ones like CSR requiring a full rebuild when updated [43]. Nevertheless, they - mainly CSR - underlie most graph analytics frameworks [19, 20, 25, 27, 50, 59, 74, 77, 78]. Recently, however, ways are being devised to develop dynamic data structures that are updateable. Most of these dynamic data structures are array-based or list-based [16, 31, 43, 64, 79, 80]. Instead of storing adjacencies as lists or arrays, [7] uses a hash table per node to store edges.

2.2 HTAP Systems

Traditionally, there used to be separate database systems dedicated to transactional and analytical workloads. The transactional (i.e. *operational*) data is periodically migrated, e.g. every night, via an offline Extract-Transform-Load (ETL) process to the analytical system for processing. This avoids resource contention between the two workloads since they are executed on separate systems. However, the periodic ETL process incurs data freshness problems as the analytics are run on an outdated version of the data. Recently, however, many application domains require analytical processing on the most recent version of the data in real-time [8, 17, 18, 24, 37, 45, 46, 82], giving rise to hybrid transactional/analytical processing (HTAP) whereby transactional and analytical workloads are executed concurrently by the same system [6, 42, 44, 46, 52, 60].

2.3 Concurrency Control

Concurrency control protocols are used to coordinate concurrent accesses to database objects by transactions, commits and aborts. We describe the relevant aspects of the Multi-Version Timestamp Ordering (MVTO) concurrency control protocol implementation in our system [39], which we extend to our delta store in order to ensure consistency during update propagation (see Section 5.3).

When a transaction starts, it is assigned a *timestamp* that uniquely identifies it. Each graph object `o` (i.e. a node or a relationship) in the main graph maintains metadata fields for concurrency control: a transaction field `txn-id` for the timestamp of the write transaction that currently locks it, a pair of begin timestamp bts and end timestamp ets that mark its access validity for read transactions, as well as a read timestamp rts for the latest transaction that read it. Below are the conditions for accessing objects by transactions.

Insert: When a transaction T with timestamp *t* adds a new graph object o, the txn-id of o is set to *t*. o remains locked by T until the end of T. The bts and ets of o are set to *t* and infinity (∞) respectively, denoting that o is visible to any transaction with a timestamp between *t* and ∞ (i.e. T and later transactions).

Update: A transaction T with timestamp *t* only updates an existing object o if it locks o (i.e. no other transaction has a lock on it already) and o has not been read by any transaction newer than T. Upon update, the ets of o is set to *t* and a newer version of o is created with bts and ets as *t* and ∞ respectively. The old version of o is unlocked for read transactions with timestamps between the old version's bts and *t*. While the new version of o remains locked until T finishes.

Read: A transaction T with timestamp *t* reads an existing object o if (1) o is not locked by any transaction and *t* lies between the bts and ets of o, or (2) o is locked by another transaction, but there is an unlocked older version of o and *t* lies between the bts and ets of the unlocked version. After reading o, the rts of o is set to *t* so that no other transaction older than T is allowed to update o.

Delete: The access condition for delete is the same as for update. The ets of o is set to *t* and a newer version of o is created with bts and ets both set to *t*. The old version of o is unlocked for read transactions with timestamps between the old version's bts and *t* while the new version remains locked until T finishes.

3 RELATED WORK

3.1 Column Stores

Column stores organize data column-wise by vertical partitioning of tables to speed up analytical queries. This way, the columns can additionally be compressed. As analytical queries scan a large number of rows but for only a few columns, columnar storage makes it possible to only fetch columns of interest. However, column stores are expensive to update in comparison to row stores. To mitigate this, column stores make use of various *delta structures* that serve as temporary stores to buffer updates before merging them into the main store. C-Store [70] maintains a read-optimized *read store* and an update-oriented *write store*. Both stores are column-oriented. MonetDB [15] is a column store that maintains the main tables as immutable, compressed columns and handles updates in delta columns. When the size of the delta columns of a table exceeds a certain fraction of the table, the deltas are merged to update the main table columns and the delta columns are emptied.

3.2 Relational HTAP

Some HTAP systems maintain single data storage for OLTP and OLAP to execute analytics on the same freshly ingested data by transactions [6]. But, it is intrinsically difficult to optimize the data organization in the storage for one without adversely affecting the other because row stores are more suited for OLTP while column stores work better for OLAP [6]. As a result, some HTAP systems e.g. L-Store [60] that use the same data layout for both OLTP and OLAP forego workload-specific optimizations of one at the expense of the other. Some compromise on data freshness and opt for hybrid layouts, e.g. [6], where *hot* tuples that are more likely to be accessed are stored in a format optimized for OLTP operations while *cold* tuples are in a format suited to OLAP requests. Moreover, the interference and resource contention between OLTP and OLAP queries as they operate on the same data degrades the overall performance [52]. As a result, other HTAP systems resort to having different storage instances - one with a row-wise layout while the other with a column-wise layout - optimized for the respective workload type. However, to maintain consistency between the data instances, updates from the OLTP part have to be propagated to the OLAP replica, which affects data freshness, unlike HTAP systems that have a single storage instance.

HTAP systems that separate storage for OLTP and OLAP, such as BatchDB [52], propagate OLTP updates to the OLAP store. In SQL Server [46], new and updated rows in its column-store indexes are first inserted into row-oriented delta stores. A column-store index can have multiple delta stores, each having a maximum of 1M rows. Delta stores that have reached the maximum capacity are merged into the column store. Some HTAP systems like SAP HANA [69] use a single main store for both OLTP and OLAP and employ a delta structure(s) for transactions while OLAP queries read both the delta(s) and the main store. RateupDB [47] has a read-only columnar *AlphaStore* and a read-write columnar *DeltaStore*. It offloads analytics to GPU.

Although the concepts of delta store in these HTAP systems (as well as the column stores discussed above) are similar to our work, however, (1) they are all on relational databases while our delta store implementation is tailored for graphs, (2) a direct conversion of the approaches in these systems to graph HTAP would result in suboptimal performance, as we show in Section 6.8, (3) the delta stores in these systems have either row or columnar layout while our delta store has a CSR-like layout (ref. Section 5.1), which is not found in any of these systems, and (4) we evaluate a persistent-memory-optimized (PMem) version of our delta store as well (ref. Section 6.5), whereas none of these systems considered persistence for their delta stores.

To the best of our knowledge, we are the first to tackle the problem of update propagation in a graph HTAP setting. In our previous work [40], we presented an initial solution for update propagation in graph HTAP while in this work, our solution handles updates faster and more efficiently and supports both static and dynamic graph data structures Furthermore, in this work, we demonstrate a strategy to enforce consistency between the main graph and the GPU-based graph replica while propagating updates, evaluate our delta store on modern hardware (PMem, in addition to DRAM), and incorporate dynamic data structures in update propagation (see Section 5 and Section 6).

3.3 Data Structures for CPU Graph Analytics

Teseo [48] and Sortledton [26] are examples of CPU-based data structures for dynamic structural graphs. These data structures are aimed at analytics on structural graphs with support for concurrent updates. Firstly, inherently, systems that employ these data structures do not make use of any delta storage since they have a single instance of the graph on CPU on which both analytics and updates are executed. Thus, they do not leverage GPU. Secondly, they incur longer analytics execution time than would otherwise be because they run analytics and updates concurrently on the same graph instance. Thirdly, these data structures are for dynamic structural graphs and thus cannot be used as property-graph stores. In Section 6.7, we show that our approach outperforms the approach of using these data structures.

4 GRAPH H^{2}TAP

4.1 System Design Considerations

Transactional graph processing entails read and update queries on nodes and relationships, as well as local neighborhood traversals. For property graphs, the nodes and relationships additionally have labels and properties, on which filter queries can be executed. These are typical for graph database systems [11] that provide support for low-latency and high-throughput ACID transactions. Meanwhile, analytics are heavily based on adjacency list scans and, as demonstrated in numerous systems dedicated to graph analytics [36], typically run on a read-only graph snapshot. However, similarly to relational databases, there is an increasing demand to support hybrid transactional and analytical graph processing [28, 84]. This has led to the emergence of graph databases like Neo4j [1], based on the property graph model.

Another dimension to graph HTAP is exploiting modern hardware, e.g. GPU, for accelerated graph analytics. The aforementioned graph HTAP maintains a single graph storage structure on the CPU for both transactional and analytical workloads, which leaves out the option of leveraging GPU for performance speedup. It is also possible for the single store to be entirely on the GPU, where not only the graph analytics but also the transaction processing is performed on the GPU, similar to GPUTx [34], a GPU-based relational OLTP engine. Aside from the performance isolation issues and limitations to workload-specific optimizations, data structures optimized for analytics on GPU target structural graphs and do not support the rich properties obtained in property graphs. This leaves us with having separate storage for transactional workloads and analytics, each residing on the computing hardware that best fits it. With heterogeneous HTAP (H^{2}TAP), CPU and GPU are essentially more suited to OLTP and OLAP tasks respectively [5, 56, 57]. Ultimately, two instances of the graph exist: the main property graph which transactions update on the CPU and the structural graph replica on GPU for analytics. Hence, updates from the main graph need to be propagated to update the GPU graph replica – whether the underlying GPU data structure is dynamic or static – so that analytics are executed on the recent version of the graph as per freshness requirements. This update propagation needs to have minimal overhead so as not to degrade the overall system performance.

4.2 Update Handling

Fig. 1 gives an overview of our delta approach to graph update handling. Our approach centers around a delta store and consists of a continuous cycle of two phases, namely the update storage phase and the update propagation phase.

Update Storage: As shown in the left part of Fig. 1, transactions update the main graph and these updates are persisted (i.e. made durable) to the graph at commit. The updates are also captured in the delta store during commit at the same time as they are persisted to the main graph. Each transaction captures its modifications to the graph as *deltas* and appends the deltas to the delta store (ref. Section 5.1). In this phase, the goal of our work is to efficiently append the deltas to the delta store with minimal overhead on the latencies of transactions and minimal memory footprint of deltas.

Update Propagation: Upon the arrival of analytics, the replica on GPU is updated using the deltas in the delta store so that the analytical queries are executed on the most recently committed graph version. For this, the delta store is first scanned (ref. Section 5.2) to prepare the deltas for updating the graph replica. This scan is independent of whether the replica is stored in a dynamic or static GPU-based graph data structure. However, the remainder of the update propagation phase depends on the underlying data structure: (1) *Dynamic Data Structures:* The deltas are coalesced and sent to the GPU all at once to amortize the transfer overhead. Once on the GPU, they are used as groups of insert and delete operations on nodes and relationships to update the dynamic data structure (top right in Fig. 1). This batched ingestion of updates is already handled by the dynamic data structure [7]. (2) *Static Data Structures:* We use CSR as a representative use case. Instead of rebuilding the CSR, which is too costly as we have shown already, we keep a copy of the CSR on the CPU and merge the deltas into it (ref. Section 5.4), resulting in a new up-to-date CSR. This new CSR is then transferred to the GPU to replace the old CSR on the GPU (bottom right in Fig. 1). In summary, when a transactional update is committed to the main graph, the process of applying the update to the graph replica starts upon the arrival of analytics. How long it will take to apply the update to the graph replica is the update propagation time.

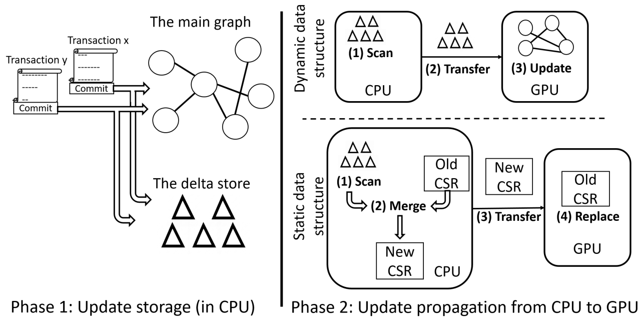


Figure 1: Update Handling.

4.3 Analytics Execution

With regard to the analytics, we assume that the graph replica fits in the GPU memory. Otherwise, techniques could be employed such as graph partitioning, multi-GPU processing, unified virtual memory, zero-copy memory access, etc. These solutions still outperform CPU-based approaches and sometimes even compete with ideal unlimited GPU memory baselines [32, 49, 55, 63].

Furthermore, at any point in time, there is only one graph replica. In other words, the analytics are executed on a single snapshot of the graph at a time, similar to relational systems like [52] and [47]. The analytics could be dispatched from a queue. Each time an analytics A arrives, it is added to the queue. When A is at the head of the queue, there are two cases: (1) *No other analytics is being executed:* In this case, A is executed if the graph replica is the recent version of the main graph as at A's arrival time. Else, the update propagation is immediately triggered with respect to the arrival time of A and A is executed after updating the replica. (2) *Another analytics B is being executed:* In this case, A is executed concurrently with B (i.e. on the same graph replica), similar to [76], if the replica is the recent version of the main graph as at the time A arrived. Otherwise, the update propagation is immediately triggered as per the arrival time of A in a pipeline fashion (while B is running). The replica is updated when B finishes and then A is executed.

5 FAST AND EFFICIENT UPDATE HANDLING

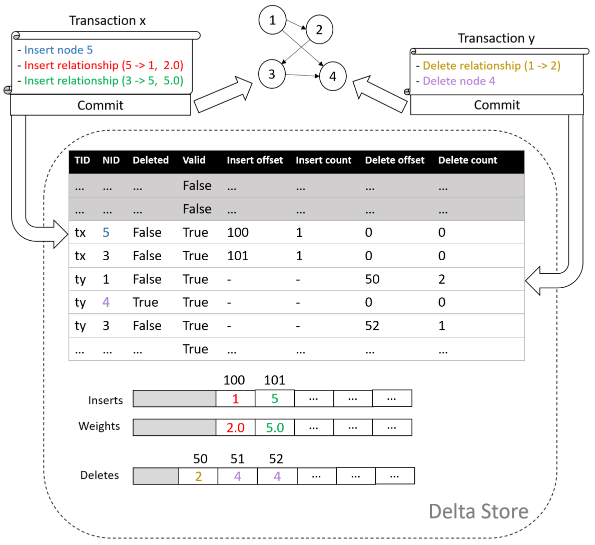
5.1 Delta Store

Our delta store buffers transactional updates on the main property graph in form of deltas in the delta store. The deltas are used to update the structural graph replica on GPU as per freshness requirements. An update transaction appends deltas to the delta store only at commit time, thereby avoiding the overhead of deleting the deltas afterwards from the delta store (as part of undo operations) if the transaction aborts. When a transaction commits, it appends one or more deltas depending on the changes it made to the main graph that alter the topology (i.e. the nodes and how they are interconnected by relationships) of the graph. Different transactional update types alter the graph topology differently and, by extension, the graph replica. These update types are node insertion, relationship insertion, node deletion and relationship deletion.

Insert/Delete Relationship: When a transaction inserts (or deletes) a relationship, (1) if the graph is a directed graph, the transaction appends a single delta that is mapped to the ID of the relationship's source node. This delta stores the ID of the destination node and the relationship weight (i.e. edge value), (2) whereas for an undirected graph, the transaction appends two deltas. One is mapped to the ID of the first node and stores the second node's ID and the relationship weight, while the other is mapped to the ID of the second node and stores the first node's ID and the relationship weight. For the remainder of this paper, we consider only directed graphs.

Insert/Delete Node: However, when a transaction inserts (or deletes) a node, a single delta is appended and mapped to the ID of the inserted (or deleted) node. For each relationship insertion associated with an inserted node by the same transaction that inserted it, the transaction appends deltas for the relationship insertion as described above. If the inserted node is a source node, the transaction stores the destination node ID and the relationship weight in the delta for the node insertion. If the inserted node is a destination node, the transaction stores the inserted node's ID (i.e. the destination node ID) and the relationship weight in a delta (created if one does not exist) associated with the source node. Similarly, if a node is deleted and is connected, as a destination node, to other nodes, the transaction appends a delta for each of those source nodes. The transaction maps each delta to the corresponding source node ID and stores the ID of the deleted node (i.e. the destination node ID) in the delta as well as the corresponding relationship weight.

Thus, each update transaction appends a separate delta for each node it updated, capturing all its modifications on that node. There is a mapping between each delta appended by a transaction and the ID of the node whose modifications the delta is capturing. That is, a delta appended by a transaction T and mapped to the ID of a node N captures all the updates made by T on N. We highlight here that a transaction appends its deltas without updating or even looking up any deltas already appended by other transactions. Thus, our delta store is an *append-only* delta store. This append-only mechanism brings the following three performance benefits: (1) It minimizes the overhead of appending deltas during the commit of each transaction because a transaction only appends its deltas without lookups on existing deltas. (2) It eliminates contention between concurrent transactions appending



TID	NID	Deleted	Valid	Insert offset	Insert count	Delete offset	Delete count
...	...	...	False	...	...	...	...
...	...	...	False	...	...	...	...
tx	5	False	True	100	1	0	0
tx	3	False	True	101	1	0	0
ty	1	False	True	-	-	50	2
ty	4	True	True	-	-	0	0
ty	3	False	True	-	-	52	1
...	...	...	True	...	...	...	...

		100	101			
Inserts		1	5	...	...	...
Weights		2.0	5.0	...	...	...

		50	51	52			
Deletes		2	4	4	...	...	...

Figure 2: Delta Store Architecture.

to the delta store because no transaction updates existing deltas. (3) It reduces the complexity of the MVTO extension to the delta store (ref. Section 5.3) because no transaction updates existing deltas during the delta store scan (ref. Section 5.2).

Delta Store Architecture: The delta store comprises a *delta table* and a set of three arrays: *inserts*, *weights* and *deletes* arrays. We implement the delta table as a linked list of fixed-sized chunks, where each chunk is an array of equally-sized objects we call *delta records*. Each delta is stored as a delta record. With this hybrid linked-list–array structure, the delta table efficiently grows to accommodate more delta records by allocating new chunks. Each delta record stores the timestamp of the transaction that appended it as well as the ID of the updated node. A delta record is uniquely identified in the delta store by a composite of its transaction timestamp and node ID. A delta record also stores a *validity* flag to indicate whether or not the delta has been used to update the graph replica (in a previous update propagation cycle) since each delta is used only once in updating the replica. This validity flag along with the transaction timestamp in delta records are used for guaranteeing consistency (see Section 5.3). Moreover, a delta record stores a *deleted* flag to indicate whether the node associated with the delta was deleted or not. This avoids storing the destination node IDs for all its outgoing relationships, which saves space since all the relationships of a deleted node are also deleted. The fields in a delta record for storing transaction timestamp, node ID, the *validity* and *deleted* flags are uniform across all delta records. However, since all delta records are equally sized but the number of captured updates varies from one delta record to another, we do not store the updates (i.e. the destination node IDs and corresponding relationship weights for inserts and deletes) directly in the delta records. Rather, we outsource them to the three arrays in the delta store: *inserts*, *weights* and *deletes*. The *inserts* array contiguously stores the destination node IDs of all inserted relationships across all delta records from all transactions. Likewise, the *weights* array contains the corresponding relationship weights (i.e. edge values) of all elements of the *inserts* array. And the *deletes* array contiguously stores the destination node IDs of all deleted relationships across all delta records from all transactions. Note that there is no need to store the corresponding relationship weights for elements in the *deletes* array because they exist already in the graph replica. This way, a delta record simply stores an integer offset to the starting positions of the destination node IDs of its inserted relationships and their corresponding relationship weights in the *inserts* and *weights* arrays respectively, as well as another integer offset to the starting position of the destination node IDs of its deleted relationships in the *deletes* array. In addition to the two integer offsets, each delta record also stores two count values to denote the number of elements that belong to it in the *inserts*, *weights* and *deletes* arrays. Thus, retrieval of a delta record's updates takes only three array lookups – a constant time complexity.

Overall, our delta store has a CSR-like layout as illustrated in Fig. 2, with all updates captured by all deltas linearized in the three arrays (analogous to the column indices and edge values arrays of a CSR) and their offsets stored in an array of delta records (analogous to the row offsets array of a CSR). The lightweight append-only design of the delta store minimizes overhead on transactions (ref. Section 6.3), its compact layout minimizes the memory footprint of the delta store itself (ref. Section 6.3) and the contiguous storing of delta elements facilitate efficient scanning of the delta store for the propagation of updates (ref. Section 6.6).
Illustrative Example: We use Fig. 2 to illustrate how transactions append deltas to the delta store. Two transactions update the sample graph. The first transaction x with timestamp t_x inserts a node with ID 5, inserts an outgoing relationship with weight 2.0 and an incoming relationship with weight 5.0 to connect the newly inserted node to nodes with IDs 1 and 3 respectively. At the same time it is committing its updates, x appends two delta records to the delta table (depicted by the third and fourth rows of the table in the figure). The first of the two delta records is mapped to the newly inserted node with ID 5, storing the ID of the destination node (i.e. 1) for its inserted outgoing relationship with weight 2.0. The destination node ID of 1 and relationship weight 2.0 are outsourced to the *inserts* and *weights* arrays respectively at index 100. Thus, the delta record stores the *inserts offset* 100 and the *inserts count* 1 (i.e. the number of inserted outgoing relationships). Similarly, the second delta record is mapped to the node with ID 3. The *validity* flags of the two delta records read *true* since they have not yet been used to update the graph replica while the *deleted* flags of the two delta records are set to *false* because the updated nodes with IDs 5 and 3 were not deleted. The second transaction y with timestamp t_y deletes the relationship connecting nodes with IDs 1 and 2, and deletes the node with ID 4. y appends a delta record for node 1 and marks a *deletes offset* of 50, i.e. the index in the *deletes* array starting from which the IDs of the destination nodes of its deleted outgoing relationships are stored. It has a *deletes count* of 2, which means two outgoing relationships of the node were deleted (i.e. the IDs at index 30 and 51). y also appends a delta record for the deleted node with ID 4. The *deleted* flag is set to *true* because the node was deleted. And because the node with ID 3 has an outgoing relationship connecting it to the deleted node, the relationship is deleted as a result. y thus appends a delta record for the node with ID 3. Note that the colours in the figure illustrate how each transactional update is reflected in the delta store.

5.2 Delta Store Scan

The arrival of analytics triggers an update propagation transaction (T_p) that begins with a delta store scan. During the scan, T_p performs a *consistency check* to identify *valid* and *visible* deltas with which to update the graph replica in the current update propagation cycle. The visibility of deltas as part of the consistency check is determined by our MVTO extension (see Section 5.3).

As discussed in Section 5.1, each updated node has an associated delta for each transaction that updated it. Multiple transactions that updated the same node would append separate deltas to capture their corresponding updates on the node. Thus, the updates associated with the same node may be spread across multiple deltas by different transactions (e.g. the two deltas mapped to the node with ID 3 in Fig. 2). T_p combines such deltas into a single delta during the scan, obviating the need for a transaction to make a check of the deltas appended by other transactions since the delta store is an append-only store, where a transaction is oblivious of the content of the deltas appended by other transactions even if they updated the same nodes. All deltas used in the current update propagation cycle are then marked as invalid by the end of the delta store scan.

5.3 Graph Consistency

On the main graph, graph consistency is guaranteed by MVTO using the metadata fields of the graph objects for concurrency control (see Section 2.3). If a transaction aborts as per MVTO, its updates will neither be persisted to the main graph nor will the transaction append deltas to the delta store. Hence, the update storage phase (i.e. appending deltas to the delta store, as discussed in Section 4.2) conforms to the MVTO implementation on the main graph, thereby guaranteeing consistency because (1) all transactions during the update storage phase are already committed transactions as per the MVTO on the main graph, and (2) the delta store is append-only and no lookups (delta record reads) are done during appending deltas. However, this is not the case during the delta store scan because (1) transactions continue appending deltas to the delta store uninterruptedly even while T_p (ref. Section 5.2) is scanning the delta store, and (2) T_p reads delta records which, unlike the main graph objects, do not have metadata fields for concurrency control.

Therefore, we extend the MVTO implementation to the delta store. This extension determines the *visibility* of deltas during the consistency check. This is because since T_p runs concurrently with update transactions appending deltas to the delta store, a transaction T that is more recent than T_p may append a delta to the delta store during the scan. T_p is not allowed to read the newly appended delta (see Section 2.3). T_p is a read-only transaction and, as a result, only the access control for read operation applies (see Section 2.3). Moreover, each delta was appended to the delta store by a transaction that has already committed, further narrowing down the access condition: that is, a delta is *visible* to T_p only if it was appended by a transaction that is older than T_p. Consequently, by identifying deltas that are both *valid* and *visible* to T_p, graph consistency between the main graph and the graph replica is enforced. With this, our delta store provides snapshot isolation. In summary, the consistency check entails the following for each delta: (1) If the delta is not *visible* to T_p, it is skipped for the current update propagation cycle. However, it will be visible for the instance of T_p in the next update propagation cycle. (2) If the delta is *visible* to T_p but *invalid*, it is skipped for the current and any subsequent update propagation cycle. (3) If the delta is both *visible* to T_p and *valid*, it is included in the current update propagation cycle.

Algorithm 1: Dynamic Graph Data Structure Update

```
1  xid ⟵ max. node ID before updates
2  foreach delta do
3      if delta.deleted then
4          deletions.add(delta.nid)
5      else if delta.nid <= xid then
6          insert_edges(delta.inserts)
7          delete_edges(delta.deletes)
8      else
9          insertions.add(delta)
10 insert_nodes(insertions)
11 delete_nodes(deletions)
```

5.4 Delta Merge

The data structure (whether static or dynamic) that stores the graph replica is transparent to our delta store. All necessary information about the updates to the main graph is captured by the deltas, which are used to update the graph replica.

Dynamic Data Structures: With a dynamic data structure, we transfer the valid and visible deltas obtained from the delta store scan to the GPU. To speed up the delta transfer, we coalesce the modifications of these deltas in the *inserts*, *weights* and *deletes* arrays of the delta store and copy them to the GPU memory all at once. We take the hash-tables-based data structure of [7] as a representative dynamic data structure (ref. Section 2.1). It supports four update operations: edge and node insertion and deletion. It implements ingesting these updates in batches. As such, once the coalesced updates captured by these deltas are transferred to the GPU, they are ingested to update the data structure. Algorithm 1 depicts a high-level illustration. For each delta, it first checks the delete flag: if the node associated with the delta is deleted, it is appended to a *deletion* queue (Line 4). If the node is newly inserted, it is appended to an *insertion* queue along with its inserted edges (Line 9). For updated nodes, the inserted edges are ingested in batches and the deleted edges likewise (Lines 6 - 7). Finally, the *insertion* and *deletion* queues are used to ingest the newly inserted nodes and remove the deleted nodes in batches respectively (Lines 10 - 11).

Static Data Structures: We use CSR as a representative of the static data structures (ref. Section 2.1). The updates captured in the deltas are in such a way that sorted column indices in the CSR are maintained. Algorithm 2 describes our proposed delta merge operation for CSR update. It divides all updated nodes into two disjoint sets based on the maximum node ID in the CSR before the update (i.e. the old CSR). The first set contains all updated nodes that existed previously in the old CSR (Line 4) while the second set contains all newly inserted nodes (Line 5). The delta merge begins with each node that existed already in the old CSR (Lines 7 - 15). If a node was not updated since the last CSR update (or for the first CSR update since the initial CSR build), the entries for the node in the new CSR is the same as in the old CSR. Note that the equality between the entries for the node in the old and new CSR here applies only to the entries in the column indices and edge values arrays. The entry for the node in the row offsets array is entirely dependent on the number of updates in the preceding nodes. Thus, the expressions `new_csr[i]` and `old_csr[i]` used in Algorithm 2 are oversimplifications of the constituent parts of a CSR. If a node was updated, however, its delta is merged with its entries in the old CSR to obtain its entries in the new CSR

Algorithm 2: Static Graph Data Structure Update

```
1  A ⟵ node IDs in all deltas
2  xid ⟵ max. node ID in old_csr
3  uid = A.upper_bound(xid)
4  L = {id | id ∈ A and id < uid}
5  U = {id | id ∈ A and id >= uid}
6  i = 0
7  foreach id ∈ L do
8      while i < id do
9          new_csr[i] = old_csr[i]
10         i++
11     new_csr[i] = merge(old_csr[i], $\text{delta}_i$)
12     i++
13 while i <= xid do
14     new_csr[i] = old_csr[i]
15     i++
16 foreach id ∈ U do
17     new_csr[i] = $\text{delta}_i$
```

(Line 11). The `merge()` function first checks the delta if the node was deleted. If that is the case, there would be an empty entry for the node in the new CSR. Else, the `merge()` function combines the *inserts* entries in the delta with the entries for the node in the old CSR and removes the *deletes* entries. The result is assigned as the entries for the node in the new CSR. Finally, the delta merge incorporates newly inserted nodes (Lines 16 - 17). Since no prior entries existed for these nodes in the old CSR, only the deltas constitute the entries for the nodes in the new CSR. Finally, the new CSR is sent to the GPU to replace the old one. Note that the new CSR also replaces the CSR copy on the CPU and serves as the old CSR in the next CSR update.

6 EVALUATION

6.1 Setup

We use two servers for our experimental evaluations:

GPU Server: Our GPU server is a dual-socket AMD EPYC 7F52 with 16 cores per socket clocked at a maximum of 3.50GHz. It is equipped with 528GB of DRAM and runs on CentOS 7.9. The machine contains a NVIDIA A100 Tensor Core GPU with 40GB of memory with a PCIe 4.0 interconnect. We use NVCC version 11.2.

PMem Server: Our PMem server is a dual-socket Intel Xeon Gold 5215 with 10 cores per socket running at 3.40 GHz max. The machine has 396GB of DRAM, 1.5TB of Intel Optane DC Persistent Memory Module (DCPMM) operating in AppDirect mode and runs on CentOS 7.9. We create an ext4 filesystem on the PMem DIMMs, mounted with the DAX option to enable direct loads and stores, thereby bypassing the OS cache. We use the Intel Persistent Memory Development Kit (PMDK) version 1.11.

6.2 Data and Workload

We use the Linked Data Benchmark Council's Social Network Benchmark (LDBC SNB) [3] datasets at *scale factors* (SF) 1, 3, 10, and 30. The LDBC-SNB [3] is based on a social network of different entity types interconnected by relationships, modelled as per the labelled property graph model. We load the data into our Poseidon [39] system as the main graph in PMem. We make use of a set of four basic update operations, each of which is centered around one of the update types that require updating the graph replica on GPU, i.e. node insertion, relationship insertion, relationship deletion and node deletion, as already discussed in Section 5.1. We describe the four operations below:

Insert Relationship: Retrieves a *Person* node with a given source node ID and a *Post* node with a given destination node ID, and creates a relationship with label *likes* to connect the two nodes.

Insert Node: Creates a new *Person* node and a new incoming relationship with label *knows* connecting the newly inserted node to an existing *Person* node in the graph.

Delete Relationship: Gets a *Person* node with a given source node ID and deletes one of its outgoing relationships.

Delete Node: Removes all the outgoing and incoming relationships that connect a *Person* node with a given ID to its neighbours and as well removes the node from the graph.

We implement and run these operations as transactional queries. The node and relationship insert operations are similar to the SNB Interactive Insert queries while the node and relationship delete operations are like the LDBC SNB Business Intelligence (BI) Delete queries [3]. Note that our focus in this evaluation is not the performance of the transactional queries themselves as they update the main graph. Rather, our focus is on evaluating the overhead of our delta approach to update handling – i.e. the update storage phase and the update propagation phase (ref. Section 4.2). For graph analytics, we use Breadth-First Search (BFS), PageRank (PR) and Single-Source Shortest Path (SSSP) algorithms from the LDBC Graphalytics benchmark on the Graph 500 dataset at scale 24 [38].

6.3 DELTA^{FE} and DELTA^{I}

We compare our fast and efficient delta approach in this paper with our previous work [40]. We denote them as `DELTA`^{FE} and `DELTA`^{I} respectively. `DELTA`^{FE} deltas are more fine-grained than `DELTA`^{I} deltas in that `DELTA`^{I} stores the adjacency lists of all updated nodes. Hence, `DELTA`^{I} stores more data in the update storage phase and, consequently, accesses more data in the update propagation phase compared to `DELTA`^{FE}. This adversely affects the performance of `DELTA`^{I} as we show below. Also, unlike `DELTA`^{FE}, `DELTA`^{I} is not suitable for dynamic data structures. And since `DELTA`^{I} only addressed static data structures (CSR), we compare the two in terms of (1) the time for executing the transactional queries, which also includes appending deltas by each transaction into the delta store during commit, (2) the memory consumption for storing the deltas in the delta store, and (3) the update propagation time, which comprises the time to scan the delta store for deltas as well as the time to merge the deltas in order to update the CSR before executing analytics.

The nodes in a graph are of varying degrees[2], depending on the degree distribution of the graph. To investigate the influence that the degrees of the updated nodes have on the performance of both `DELTA`^{I} and `DELTA`^{FE}, we sort the node IDs based on the degrees of the nodes (i.e. the sizes of their adjacency lists) and define a *window* of IDs for updates. We *slide* the *window* to select nodes with low degrees (denoted as `LoDeg` in the plots) or nodes with high degrees (denoted as `HiDeg` in the plots) to be updated by the queries.

Transactional Update Time: Fig. 3 shows the transactional update time comparison between `DELTA`^{I} and `DELTA`^{FE} for both high and low node degrees using the SF 1 data. The execution of the insert relationship, insert node, delete relationship and delete node queries are distributed as 66%, 22%, 11% and 1% of the total number of queries respectively. We conduct an evaluation of the four update types separately, followed by a mixed workload containing all the update types. Based on Fig. 3, we come to the following conclusions.

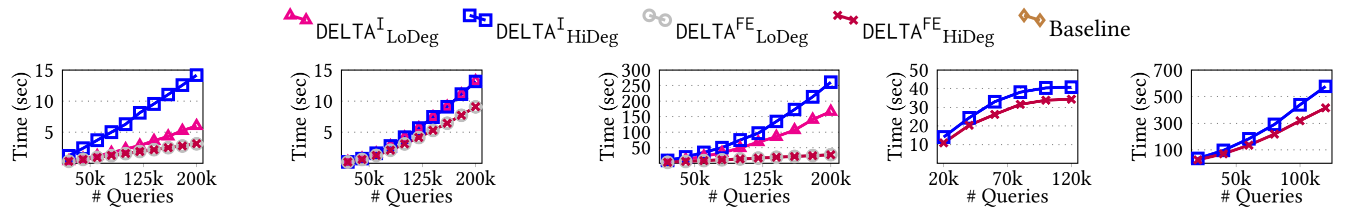
Firstly, `DELTA`^{FE} has a higher performance than `DELTA`^{I} in all cases, especially with a larger number of queries. Secondly, unlike `DELTA`^{I}, `DELTA`^{FE} exhibits the same performance for both low- and high-degree nodes. In other words, the `DELTA`^{I} is not scalable with increasing node degrees while `DELTA`^{FE} is not even influenced by the degrees of the particular nodes being updated. Thus, we see the performance difference between `DELTA`^{FE} and `DELTA`^{I} being even higher when the updated nodes are high-degree nodes than when they are low-degree nodes.

It is worth explaining Fig. 3 further. (1) The performance difference between `DELTA`^{FE} and `DELTA`^{I} is much more for insert relationship than other updates because repeatedly inserting relationships increases the degree of the updated nodes. And the higher the degree of the nodes, the bigger the performance difference between `DELTA`^{FE} and `DELTA`^{I}. (2) The performance of `DELTA`^{I} for delete node with low node degrees is the same as with high node degrees. The reason is that the appended deltas for the deleted nodes are all empty since there are no more connected relationships remaining for any node after it is deleted (see Section 6.2), irrespective of the degree of the node before it was deleted. (3) We only evaluate delete relationship for high node degrees because deleting relationships is limited to the total number of outgoing relationships attached to the *Person* nodes selected by the *window* for updating (see Section 6.2), even if the nodes may still have incoming relationships. Similarly, we only evaluate the mixed workload for high-degree nodes because of the delete relationship operations contained therein.

We compare the transactional update times of both `DELTA`^{I} and `DELTA`^{FE} (i.e. plus the overhead of appending deltas) with the baseline transactional update time. By baseline here, we mean the default execution of transactional updates without any delta mechanism (i.e. without the overhead of appending deltas). The update times for the baseline and `DELTA`^{FE} are close to each other in all cases, as shown in Fig. 6. To further validate this on a larger-sized graph, we execute the mixed workload on the SF 10 data and plot the transactional update times of the baseline and `DELTA`^{FE} in Fig. 8. Similarly to Fig. 6, we see from Fig. 8 that the update times are similar. As such, there is no correlation between the appended deltas and the transactional update time of `DELTA`^{FE}. On the other hand, there is a difference between the update time of the baseline and that of `DELTA`^{I}, essentially the same as the difference between `DELTA`^{I} and `DELTA`^{FE} in Fig. 3. This delta append overhead of `DELTA`^{I} compared to the baseline is directly correlated to the total size of deltas as depicted in Fig. 7. The overhead exists only for `DELTA`^{I} but not for `DELTA`^{FE} because, as Fig. 4 shows, `DELTA`^{FE} has orders of magnitude less storage overhead than `DELTA`^{I} to such an extent that `DELTA`^{FE} delta append overhead is negligible and does not influence the overall transactional update time.

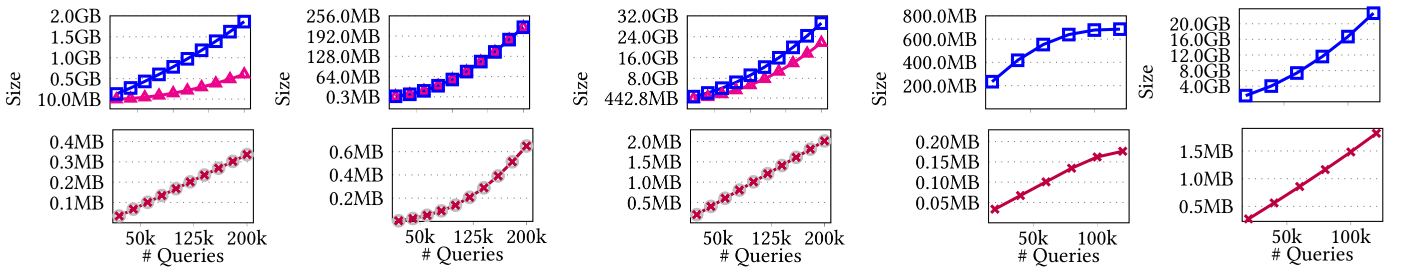
Delta Memory Footprint: Next, we evaluate `DELTA`^{I} and `DELTA`^{FE} in terms of their memory footprint i.e. the total size of deltas in the delta store. The size of each delta depends on the number of modifications it captures, which are stored in the *inserts*, *weights* and *deletes* arrays in the delta store, as explained in Section 5.1. And these modifications vary from one delta to another. The delta sizes are therefore the total size of the stored array elements in the *inserts*, *weights* and *deletes* arrays. Each array element in the

[2]the degree of a node is the number of relationships connected to that node



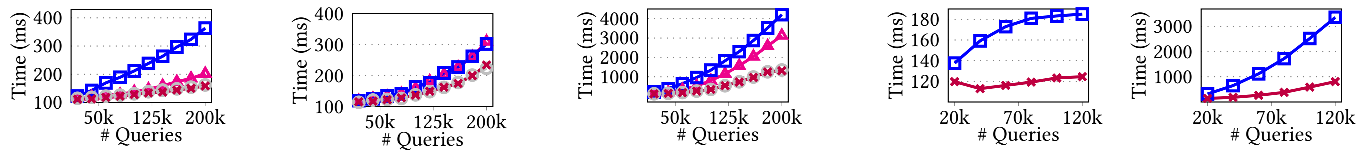
(a) Insert Node (b) Delete Node (c) Insert Relationship (d) Delete Relationship (e) Mixed

Figure 3: Transactional Update Time.



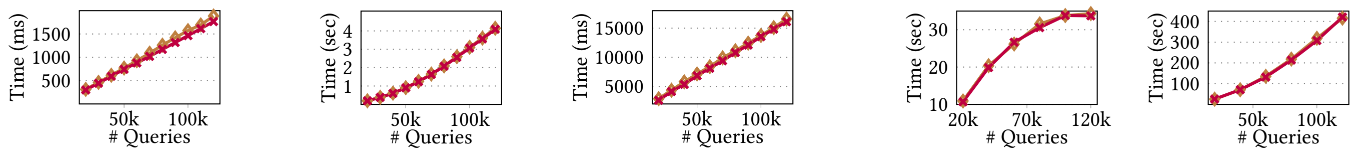
(a) Insert Node (b) Delete Node (c) Insert Relationship (d) Delete Relationship (e) Mixed

Figure 4: Delta Memory Footprint.



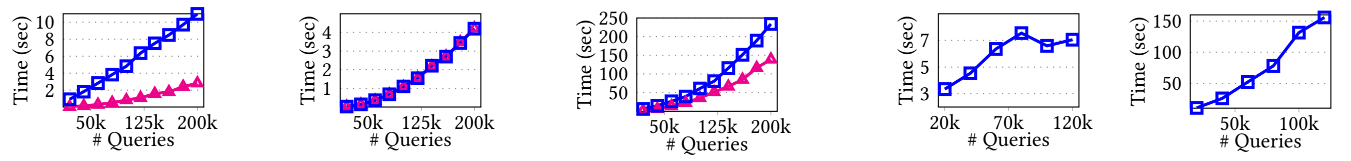
(a) Insert Node (b) Delete Node (c) Insert Relationship (d) Delete Relationship (e) Mixed

Figure 5: Update Propagation Time.



(a) Insert Node (b) Delete Node (c) Insert Relationship (d) Delete Relationship (e) Mixed

Figure 6: Transactional Update Time (Baseline vs $\texttt{DELTA}^{\texttt{FE}}_{\text{HiDeg}}$ on SF 1).



(a) Add Node (b) Delete Node (c) Add Relationship (d) Delete Relationship (e) Mixed

Figure 7: Delta Append Overhead.

inserts and *deletes* arrays is an 8-byte node ID while in the *weights* array, each element is an 8-byte relationship weight.

As Fig. 4 shows, $\texttt{DELTA}^{\texttt{FE}}$ consumes less memory compared to $\texttt{DELTA}^{\texttt{I}}$ by orders of magnitude (hence the reason we use different scales in the plots). This is expected since $\texttt{DELTA}^{\texttt{I}}$ not only stores the entire state of the adjacency list of each updated node but also for each transaction that updated the node. This results in the transactional update time difference between $\texttt{DELTA}^{\texttt{I}}$ and $\texttt{DELTA}^{\texttt{FE}}$, especially for a large number of updates and/or high-degree nodes (see Fig. 3). Fig. 4 also shows that the memory

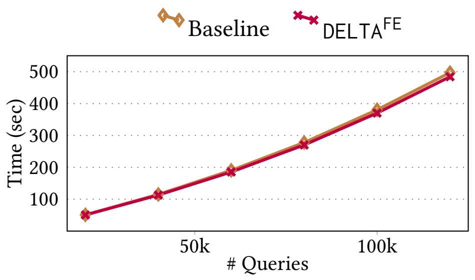


Figure 8: Transactional Update Time (Baseline vs DELTA^{FE}$_{HiDeg}$ on SF 10).

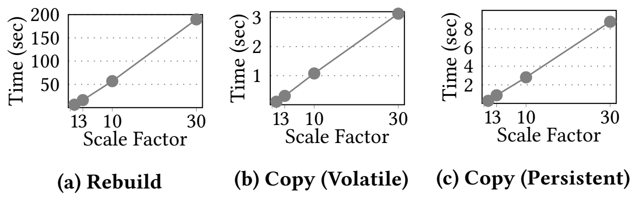


Figure 9: CSR Rebuild and CSR Copy.

footprint of DELTA^{FE} is the same regardless of the degrees of the updated nodes, thus validating that DELTA^{FE} is much more scalable than DELTA^{I} with higher degrees of nodes. And this is particularly crucial in settings where the memory budget is limited. For delete node, however, the memory footprint of the DELTA^{I} is the same for low- and high-degree nodes since the deltas for deleted nodes are empty, as explained earlier. Nonetheless, even for the delete node, the memory footprint of DELTA^{I} is much higher than that of DELTA^{FE}.

Update Propagation Time: Furthermore, we analyze the time for propagating updates to the CSR so as to execute analytics on an updated version of the graph. Precisely, we compare between DELTA^{I} and DELTA^{FE} based on the combined time to scan the delta store for consistency check (ref. Section 5.2 and Section 5.3) and time of the merge operation for updating the CSR (ref. Section 5.4). From Fig. 5, we draw similar conclusions to those from Fig. 3. The DELTA^{FE} is faster than the DELTA^{I} in propagating updates in all cases. This is more so with a larger number of queries, i.e. larger number of deltas. Also, the DELTA^{I} performs worse with high-degree nodes than with low-degree nodes while the DELTA^{FE} is not affected by the degrees of the updated nodes, making it not only faster with increasing node degrees, but also more scalable. We discuss the update propagation times in more detail in Section 6.4.

6.4 CSR Rebuild and CSR Update

We further evaluate DELTA^{FE} with regards to update propagation. A straightforward way of handling updates and updating a CSR is to simply rebuild the CSR each time the main graph gets updated. However, the overhead of constantly rebuilding the CSR is quite expensive especially for large graphs where the rebuild time is even much higher than the actual execution time of the analytics, as we showed previously. We plot the CSR rebuild times for graphs of different SNB scale factors in Fig. 9 to illustrate how the rebuild time grows for larger graph sizes. In Section 6.3, we have shown that unlike DELTA^{I}, the transactional update time of DELTA^{FE} is not correlated with the appended deltas. This leaves

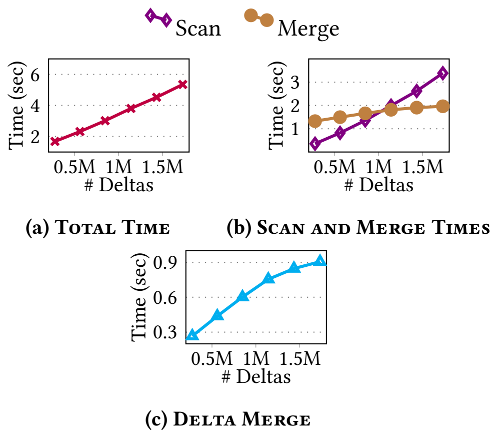


Figure 10: Update Propagation Time (Detailed).

us with the update propagation time as the overhead of our delta approach using DELTA^{FE}. Fig. 10a shows the correlation between the number of appended deltas and the update propagation time for the SF 10 data. Comparing with the CSR rebuild time of SF 10 graph, i.e. 56921.64ms, we see that our delta approach is faster than constantly rebuilding the CSR. However, we can see from the figure that the overhead of the delta approach increases with an increasing number of deltas and, thus, could be higher than the CSR rebuild time depending on the number of appended deltas. Therefore, we propose a cost model to decide when to use which of the two approaches.

We further break down the update propagation time into its constituent time components, i.e. the delta store scan time and the merge operation time. Fig. 10b shows the correlation between each of the two and the number of appended deltas. The delta store scan is more strongly correlated with the number of deltas than the merge operation time and eventually becomes the dominant part of the update propagation time. The merge operation times lie within a limited time range. Since the merge operation comprises copying parts of the CSR for the unchanged nodes (Lines 9 and 14 of Algorithm 2) and modifying parts of the CSR for the updated and the newly inserted nodes (Lines 11 and 17 of Algorithm 2), comparing the merge times with the time to simply copy the entire CSR (1075.94ms for SF 10 as depicted in Fig. 9b) reveals that the merge operation is dominated by the copying (which depends on the size of the graph) rather than the modifying (which depends on the number of the deltas). This is because the deltas are only a fraction in comparison to the entire graph. We plot the CSR copy times for different scale factors in Fig. 9b to show how the copy time grows with larger graph sizes.

Thus, our cost model constitutes (1) a model for the correlation between the number of appended deltas and the time for scanning the delta store, (ref. Fig. 10b), (2) a model for the correlation between the CSR copying part of the merge operation time and the graph size (ref. Fig. 9b), (3) a model for the correlation between the CSR modifying part of the merge operation time and the number of deltas (ref. Fig. 10c), and (4) a model for the correlation between the CSR rebuild time and the graph size (ref. Fig. 9a). The time of delta store scan, the merge operation and the CSR rebuild are obtainable from the mathematical model extractions of the model descriptions above. The cost model compares the rebuild time overhead and the delta time overhead (i.e. delta store scan and merge operation) to decide between the two approaches.

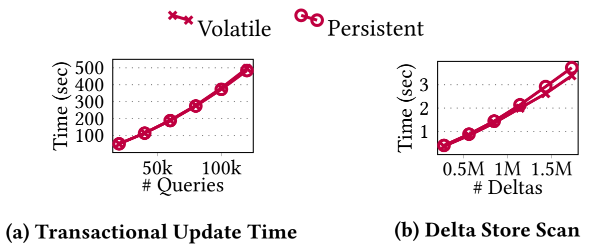


Figure 11: Volatile vs Persistent Delta Store.

From the cost model, we get a delta size threshold, which is the minimum number of deltas for which the rebuild overhead is less than the delta overhead. We incorporate this threshold into our delta store such that before any transaction appends its deltas into the delta store at commit, it checks to ensure that the number of deltas presently in the delta store plus the number of its deltas does not exceed the threshold. If that is the case, it appends its deltas to the delta store. Otherwise, it does not append its deltas but sets OFF a *delta mode* flag in the delta store (which is ON by default). Any subsequent update transaction that finds the *delta mode* flag switched OFF also does not append any deltas. At this point, the delta store is cleared at once to remove all deltas but the *delta mode* flag stays OFF. Hence, the CSR is rebuilt upon the next execution of analytics. As soon as the CSR is rebuilt, the *delta mode* flag is switched back ON and all subsequent update transactions append their deltas into the delta store onwards. We leave a detailed evaluation of our cost model for future work.

6.5 Volatile and Persistent Delta Store

The delta store is lost in case of a system crash or restart because it is volatile. The CSR is also lost and would have to be rebuilt from the main graph However, similar to the main graph on PMem, the delta store can be instantly recovered if it is also stored on PMem. Such a persistent delta store instantly continues to serve its purpose upon recovery due to failure or restart. This is an advantage, particularly in cases where there are frequent system failures or restarts.

We implement a PMem-based DELTA^{FE} while employing some of the PMem optimizations used in Poseidon's main storage [39] in order that the persistent delta store achieves a near-DRAM performance. We use the SF 10 data. Fig. 11a shows that the transactional update times with the volatile and persistent delta stores are close to each other. This is as expected for two main reasons. Firstly, the overhead of appending deltas with DELTA^{FE} does not influence the update time as discussed in Section 6.3 (see Fig. 6). Secondly, the PMem optimizations have been proven to achieve close-to-DRAM performance for transactional graph processing [39]. From Fig. 11b, we also see that the time of scanning the persistent delta store is close to that of scanning the volatile delta store. Note that with a persistent delta store, only appending deltas to the delta store and scanning the delta store involve PMem. Thus, we do not compare the volatile and persistent delta stores for the merge operation.

To recover an up-to-date CSR from the persistent delta store, since the delta store builds upon the current CSR, there is also a need for a persistent copy of the CSR in addition to the default volatile CSR. This persistent CSR copy is only used for recovery purposes (not for analytics) and is overwritten by the contents of the default volatile CSR each time the latter is updated with deltas. There is a memory and performance overhead to this, both of which depend on the size of the graph. The memory overhead is the extra copy of the CSR on PMem, which is negligible since PMem has a much higher capacity than DRAM. The performance overhead is in keeping the persistent CSR copy up-to-date, each time the default volatile CSR is updated. We measure this performance overhead for the SF 10 graph as 2805.06ms (see Fig. 9c). Thus, the persistent delta store carries this extra 2805.06ms overhead in each update propagation cycle. However, this overhead is easily avoidable because the execution of analytics does not have to wait for it since as mentioned previously, the persistent CSR is only for recovery purposes and not for analytics.

Nonetheless, as Fig. 11 and Fig. 10 show, the delta approach using a persistent delta store in PMem is also faster than the CSR rebuild, similarly to the volatile delta store, when the number of deltas does not exceed a certain size threshold. The corresponding size threshold for the persistent delta store could also be obtained from our cost model by incorporating the factors peculiar to PMem as discussed above. We also leave this for future work.

6.6 Update Handling

We evaluate DELTA^{FE} with respect to update handling, i.e. the update storage phase and the update propagation phase combined, using the SF 10 data on our GPU server for both static and dynamic data structures. The storage medium for the main graph here is disk not PMem, unlike on our PMem server as used in the previous evaluations. One of the advantages of having the main graph on PMem is that no loading of the graph is required from storage to memory since PMem is also byte-addressable. Also, with PMem, recovery is instant. Nevertheless, on the GPU server, we ignore the disk access latency in fetching the graph data from disk and the additional overhead of serialization. For simplicity, we assume that the main graph is already in DRAM.

We refer to Fig. 1. The update storage phase (shown on the left of the figure) consists of appending deltas to the delta store. We execute update transactions as to append 2014347 (approx. 2M) deltas using DELTA^{FE}. We further run the same update transactions but without appending any deltas (i.e. the baseline updates). The difference is only 1068.13ms. This corroborates our previous evaluation which shows that there is no correlation between the appended deltas and the transactional update time with DELTA^{FE}, unlike DELTA^{I} (ref. Section 6.3). Next is the update propagation phase (on the right of the figure). We have shown in Section 6.4 that how long it takes in the update propagation phase to apply the main graph updates to the graph replica (ref. Section 4.2) is correlated to the number of appended deltas. The update propagation phase begins with scanning the delta store. The time to scan the delta store for the 2M deltas is 2595.95ms. As our evaluations in Section 6.4 show, there is a correlation between this delta store scan time and the number of appended deltas. The remainder of the update propagation phase has two cases:

Dynamic Data Structures: For dynamic data structures, we coalesce the modifications of the *valid* and *visible* deltas in the *inserts*, *weights* and *deletes* arrays and copy them to GPU in a single transfer (shown on the top right in Fig. 1). The transfer overhead is a mere 4.75ms. This totals the update propagation time for the modifications on the main graph captured by the 2M deltas to update the graph replica on GPU at 2600.7ms.

Static Data Structures: As for CSR, the default solution would be to rebuild the CSR and transfer it to the GPU (bottom right of Fig. 1). Rebuilding the CSR from the SF 10 graph takes 33133.51ms while copying the CSR to GPU takes 720.64ms. We reduce this

		BFS	PR	SSSP
Sortledton	Analytics on CPU	**1.48**	**21.34**	**57.30**
DELTA^{FE}	Update Propagation	5.38	5.38	5.38
	Analytics on GPU	0.07	0.30	0.13
	Sum	**5.45**	**5.68**	**5.51**

Table 1: HTAP and H^{2}TAP Analytics Latency (in seconds).

huge CSR rebuild bottleneck by devising a way to update the CSR via merging of the deltas with the CSR and sending the updated CSR to GPU. The merge operation for the 2M deltas takes 2064.44ms. Similarly, our evaluations in the previous section show that the merge time is correlated to the number of deltas. The update propagation time is 4660.39ms. Thus, for 2M deltas, our delta reduces the overhead of CSR rebuild by 85%.

6.7 HTAP and H^{2}TAP

We compare DELTA^{FE} with the approach that utilizes the CPU-based data structures for dynamic structural graph analytics discussed in Section 3.3. As mentioned already, by design, the latter approach foregoes leveraging GPU for accelerated analytics. We select Sortledton [26] for the comparison because, so far, it is the best-performing in this category of CPU data structures. We compare between (a) analytics execution time of Sortledton on CPU and (b) the sum of the update propagation time in DELTA^{FE} and the analytics execution time on GPU. Since the update propagation time correlates with the number of deltas (see Section 6.5), we evaluate 2M deltas in order to stress DELTA^{FE}. Note that with Sortledton, the analytics are executed on CPU concurrently with updates, and as a result, this approach incurs extra performance penalties due to a lack of performance isolation.

Table 1 shows the execution times for BFS, PR and SSSP on the Graph 500 data at scale 24. We see that DELTA^{FE} (Line 4) significantly outperforms Sortledton (Line 1) in PR and SSSP, as a result of the fast analytics execution on GPU. Note that results from prior works have also shown that GPU achieves up to orders of magnitude faster analytics execution than CPU [25, 78]. The update propagation is the dominant latency factor in DELTA^{FE} because of the huge number of deltas. If the analytics is compute-heavy, the performance gains from GPU will be higher than the update propagation time, as can be seen with PR and SSSP. Otherwise, the GPU acceleration does not pay off the update propagation time as is the case for BFS, which is not as compute-heavy [67]. Furthermore, we compare between DELTA^{FE} and Sortledton as to the total latency for BFS, PR and SSSP, assuming all three algorithms are dispatched for execution. We take two scenarios: (1) all three analytics arrive at the same time, and (2) an analytics arrives after other finishes. With regards to the first scenario, the total latency is the latency of the longest-running analytics, which is 57.30s in Sortledton and 5.68s in DELTA^{FE}. As for the second scenario, the total latency is the sum of the latencies of all analytics, i.e. 80.12s in Sortledton and 16.64s in DELTA^{FE}.

In summary, the efficiency of DELTA^{FE} over Sortledton increases (1) when analytics are compute-heavy (i.e. relatively long-running), which is the typical case for graph analytics [64] like PR and SSSP, (2) when several analytics are executed on the same graph replica version e.g. as a batch. Since the graph replica needs to be updated only once, this amortizes the update propagation time across the analytics even if they are not compute-heavy. And (3) when the updates between successive arrival of analytics are not huge.

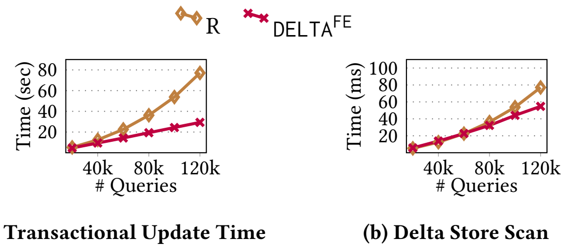


Figure 12: DELTA^{FE} vs Relational Approaches.

6.8 Delta Store Optimizations

Lastly, we evaluate the optimizations in DELTA^{FE} compared to a direct conversion of delta store approaches in relational systems like [47] to graphs. These systems store all the columns in their delta stores. Thus, a direct conversion would result in a delta store that stores full graph objects with complete MVCC information, thereby increasing the delta store size and the update propagation overhead. Furthermore, their delta store entries are updateable. As a result, a direct conversion would incur additional overhead in lookups during transaction commits. Altogether, a direct conversion of these approaches leads to suboptimal performance as we stated in Section 3.2. Fig. 12 depicts the performance difference between DELTA^{FE} and our implementation of a direct conversion of the aforementioned relational approaches (denoted as R), thereby showing the performance gains resulting from the graph-aware optimizations in DELTA^{FE}, as presented in Section 5.

7 CONCLUSION

In this paper, we presented an approach to handling updates in graph H^{2}TAP, based on a delta store implementation tailored for graphs. Transactional updates on a main graph are captured by way of deltas in the delta store and subsequently propagated to update the replica of the graph on GPU. Our delta approach is fast and efficient in terms of update storage and update propagation. Furthermore, we introduce a mechanism for enforcing consistency between the main graph and the graph replica over the course of the update propagation. We conduct extensive experimental evaluations of our delta approach and show that it is significantly better than the existing work with respect to performance and memory footprint. Our evaluations also show that our approach outperforms both CPU-only graph solutions and solutions based on a direct conversion of existing approaches in relational systems to graphs. Moreover, in addition to DRAM, we show the viability of our delta store in persistent memory.

ACKNOWLEDGMENTS

This work was partially funded by the German Research Foundation (DFG) in the context of the project “Processing-In-Memory Primitives for Data Management (PIMPMe)” (SA 782/31) as part of the priority program “Disruptive Memory Technologies” (SPP 2377), “Hybrid Transactional/Analytical Graph Processing in Modern Memory Hierarchies (#TAG)” (SA 782/28-2) as part of the priority program “Scalable Data Management for Future Hardware” (SPP 2037), the Carl-Zeiss-Stiftung under the project “Memristive Materials for Neuromorphic Electronics (MemWerk)” and the Thuringian Ministry for Economy, Science and Digital Society under the project “thurAI”.

REFERENCES

[1] [n.d.]. Neo4j. https://neo4j.com/. Retrieved: February 17, 2023.

[2] Waqas Ali, Muhammad Saleem, Bin Yao, Aidan Hogan, and Axel-Cyrille Ngonga Ngomo. 2022. A survey of RDF stores & SPARQL engines for querying knowledge graphs. *VLDB J.* 31, 3 (2022), 1–26. https://doi.org/10.1007/s00778-021-00711-3

[3] Renzo Angles, János Benjamin Antal, Alex Averbuch, Peter A. Boncz, Orri Erling, Andrey Gubichev, Vlad Haprian, Moritz Kaufmann, Josep Lluís Larriba-Pey, Norbert Martínez-Bazan, József Marton, Marcus Paradies, Minh-Duc Pham, Arnau Prat-Pérez, Mirko Spasic, Benjamin A. Steer, Gábor Szárnyas, and Jack Waudby. 2020. The LDBC Social Network Benchmark. *CoRR* abs/2001.02299 (2020). arXiv:2001.02299 http://arxiv.org/abs/2001.02299

[4] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan L. Reutter, and Domagoj Vrgoc. 2017. Foundations of Modern Query Languages for Graph Databases. *ACM Comput. Surv.* 50, 5 (2017), 68:1–68:40. https://doi.org/10.1145/3104031

[5] Raja Appuswamy, Manos Karpathiotakis, Danica Porobic, and Anastasia Ailamaki. 2017. The Case For Heterogeneous HTAP. In *CIDR 2017*. http://cidrdb.org/cidr2017/papers/p21-appuswamy-cidr17.pdf

[6] Joy Arulraj, Andrew Pavlo, and Prashanth Menon. 2016. Bridging the Archipelago between Row-Stores and Column-Stores for Hybrid Workloads. In *SIGMOD 2016*. ACM, 583–598. https://doi.org/10.1145/2882903.2915231

[7] Muhammad A. Awad, Saman Ashkiani, Serban D. Porumbescu, and John D. Owens. 2020. Dynamic Graphs on the GPU. In *IPDPS 2020*. IEEE, 739–748. https://doi.org/10.1109/IPDPS47924.2020.00081

[8] Ronald Barber, Adam J. Storm, Yuanyuan Tian, Pinar Tözün, Yingjun Wu, Christian Garcia-Arellano, Ronen Grosman, Guy M. Lohman, C. Mohan, René Müller, Hamid Pirahesh, Vijayshankar Raman, and Richard Sidle. 2019. WiSer: A Highly Available HTAP DBMS for IoT Applications. In *IEEE BigData 2019*. 268–277. https://doi.org/10.1109/BigData47090.2019.9006519

[9] Jiri Barnat, Petr Bauch, Lubos Brim, and Milan Ceska. 2011. Computing Strongly Connected Components in Parallel on CUDA. In *IPDPS 2011*. IEEE, 544–555. https://doi.org/10.1109/IPDPS.2011.59

[10] Nathan Bell and Michael Garland. 2009. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *SC 2009*. ACM. https://doi.org/10.1145/1654059.1654078

[11] Maciej Besta, Emanuel Peter, Robert Gerstenberger, Marc Fischer, Michal Podstawski, Claude Barthels, Gustavo Alonso, and Torsten Hoefler. 2019. Demystifying Graph Databases: Analysis and Taxonomy of Data Organization, System Designs, and Graph Queries. *CoRR* abs/1910.09017 (2019). arXiv:1910.09017 http://arxiv.org/abs/1910.09017

[12] Mauro Bisson and Massimiliano Fatica. 2018. Update on Static Graph Challenge on GPU. In *HPEC 2018*. IEEE, 1–8. https://doi.org/10.1109/HPEC.2018.8547514

[13] Mauro Bisson, Everett H. Phillips, and Massimiliano Fatica. 2016. A CUDA implementation of the pagerank pipeline benchmark. In *HPEC 2016*. IEEE, 1–7. https://doi.org/10.1109/HPEC.2016.7761620

[14] Thorsten Blaß and Michael Philippsen. 2019. Which Graph Representation to Select for Static Graph-Algorithms on a CUDA-capable GPU. In *GPGPU@ASPLOS 2019*. ACM, 22–31. https://doi.org/10.1145/3300053.3319416

[15] Peter A. Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR 2005*. 225–237. http://cidrdb.org/cidr2005/papers/P19.pdf

[16] Federico Busato, Oded Green, Nicola Bombieri, and David A. Bader. 2018. Hornet: An Efficient Data Structure for Dynamic Sparse Graphs and Matrices on GPUs. In *HPEC 2018*. IEEE, 1–7. https://doi.org/10.1109/HPEC.2018.8547541

[17] Shaosheng Cao, Xinxing Yang, Cen Chen, Jun Zhou, Xiaolong Li, and Yuan Qi. 2019. TitAnt: Online Real-time Transaction Fraud Detection in Ant Financial. *Proc. VLDB Endow.* 12, 12 (2019), 2082–2093. https://doi.org/10.14778/3352063.3352126

[18] Biswapesh Chattopadhyay, Priyam Dutta, Weiran Liu, Ott Tinn, Andrew McCormick, Aniket Mokashi, Paul Harvey, Hector Gonzalez, David Lomax, Sagar Mittal, Roee Ebenstein, Nikita Mikhaylin, Hung-Ching Lee, Xiaoyan Zhao, Tony Xu, Luis Perez, Farhad Shahmohammadi, Tran Bui, Neil Mckay, Selcuk Aya, Vera Lychagina, and Brett Elliott. 2019. Procella: Unifying serving and analytical data at YouTube. *Proc. VLDB Endow.* 12, 12 (2019), 2022–2034. https://doi.org/10.14778/3352063.3352121

[19] Shuai Che. 2014. GasCL: A vertex-centric graph model for GPUs. In *HPEC 2014*. IEEE, 1–6. https://doi.org/10.1109/HPEC.2014.7040962

[20] Shuai Che, Bradford M. Beckmann, and Steven K. Reinhardt. 2014. BelRed: Constructing GPGPU graph applications with software building blocks. In *HPEC 2014*. IEEE, 1–6. https://doi.org/10.1109/HPEC.2014.7040961

[21] Andrew A. Davidson, Sean Baxter, Michael Garland, and John D. Owens. 2014. Work-Efficient Parallel GPU Methods for Single-Source Shortest Paths. In *IPDPS 2014*. IEEE, 349–359. https://doi.org/10.1109/IPDPS.2014.45

[22] Shrinivas Devshatwar, Madhur Amilkanthwar, and Rupesh Nasre. 2016. GPU centric extensions for parallel strongly connected components computation. In *GPGPU@PPoPP 2016*. ACM, 2–11. https://doi.org/10.1145/2884045.2884048

[23] Orri Erling, Alex Averbuch, Josep Lluís Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat-Pérez, Minh-Duc Pham, and Peter A. Boncz. 2015. The LDBC Social Network Benchmark: Interactive Workload. In *SIGMOD 2015*. ACM, 619–630. https://doi.org/10.1145/2723372.2742786

[24] Yupeng Fu and Chinmay Soman. 2021. Real-time Data Infrastructure at Uber. In *SIGMOD 2021*. ACM, 2503–2516. https://doi.org/10.1145/3448016.3457552

[25] Zhisong Fu, Bryan B. Thompson, and Michael Personick. 2014. MapGraph: A High Level API for Fast Development of High Performance Graph Analytics on GPUs. In *GRADES@SIGMOD/PODS 2014*. CWI/ACM, 2:1–2:6. https://doi.org/10.1145/2621934.2621936

[26] Per Fuchs, Jana Giceva, and Domagoj Margan. 2022. Sortledton: a universal, transactional graph data structure. *Proc. VLDB Endow.* 15, 6 (2022), 1173–1186. https://www.vldb.org/pvldb/vol15/p1173-fuchs.pdf

[27] Abdullah Gharaibeh, Lauro Beltrão Costa, Elizeu Santos-Neto, and Matei Ripeanu. 2012. A yoke of oxen and a thousand chickens for heavy lifting graph processing. In *PACT 2012*. ACM, 345–354. https://doi.org/10.1145/2370816.2370866

[28] Jana Giceva and Mohammad Sadoghi. 2019. Hybrid OLTP and OLAP. In *Encyclopedia of Big Data Technologies*. Springer. https://doi.org/10.1007/978-3-319-63962-8_179-1

[29] John R. Gilbert, Steven P. Reinhardt, and Viral B. Shah. 2006. High-Performance Graph Algorithms from Parallel Sparse Matrices. In *PARA 2006*. Springer, 260–269. https://doi.org/10.1007/978-3-540-75755-9_32

[30] Joseph L. Greathouse and Mayank Daga. 2014. Efficient Sparse Matrix-Vector Multiplication on GPUs Using the CSR Storage Format. In *SC 2014*. IEEE, 769–780. https://doi.org/10.1109/SC.2014.68

[31] Oded Green and David A. Bader. 2016. cuSTINGER: Supporting dynamic graph algorithms for GPUs. In *HPEC 2016*. IEEE, 1–6. https://doi.org/10.1109/HPEC.2016.7761622

[32] Wei Han, Daniel Mawhirter, Bo Wu, and Matthew Buland. 2017. Graphie: Large-Scale Asynchronous Graph Traversals on Just a GPU. In *PACT 2017*. IEEE, 233–245. https://doi.org/10.1109/PACT.2017.41

[33] Pawan Harish and P. J. Narayanan. 2007. Accelerating Large Graph Algorithms on the GPU Using CUDA. In *HiPC 2007*. Springer, 197–208. https://doi.org/10.1007/978-3-540-77220-0_21

[34] Bingsheng He and Jeffrey Xu Yu. 2011. High-throughput transaction executions on graphics processors. *Proc. VLDB Endow.* 4, 5 (2011), 314–325. https://doi.org/10.14778/1952376.1952381

[35] Tim Hegeman and Alexandru Iosup. 2018. Survey of Graph Analysis Applications. *CoRR* abs/1807.00382 (2018). arXiv:1807.00382 http://arxiv.org/abs/1807.00382

[36] Safiollah Heidari, Yogesh Simmhan, Rodrigo N. Calheiros, and Rajkumar Buyya. 2018. Scalable Graph Processing Frameworks: A Taxonomy and Open Challenges. *ACM Comput. Surv.* 51, 3 (2018), 60:1–60:53. https://doi.org/10.1145/3199523

[37] Yanxiang Huang, Bin Cui, Jie Jiang, Kunqian Hong, Wenyu Zhang, and Yiran Xie. 2016. Real-time Video Recommendation Exploration. In *SIGMOD 2016*. ACM, 35–46. https://doi.org/10.1145/2882903.2903743

[38] Alexandru Iosup, Tim Hegeman, Wing Lung Ngai, Stijn Heldens, Arnau Prat-Pérez, Thomas Manhardt, Hassan Chafi, Mihai Capota, Narayanan Sundaram, Michael J. Anderson, Ilie Gabriel Tanase, Yinglong Xia, Lifeng Nai, and Peter A. Boncz. 2016. LDBC Graphalytics: A Benchmark for Large-Scale Graph Analysis on Parallel and Distributed Platforms. *Proc. VLDB Endow.* 9, 13 (2016), 1317–1328. https://doi.org/10.14778/3007263.3007270

[39] Muhammad Attahir Jibril, Alexander Baumstark, Philipp Götze, and Kai-Uwe Sattler. 2021. JIT happens: Transactional Graph Processing in Persistent Memory meets Just-In-Time Compilation. In *EDBT 2021*. OpenProceedings.org, 37–48. https://doi.org/10.5441/002/edbt.2021.05

[40] Muhammad Attahir Jibril, Alexander Baumstark, and Kai-Uwe Sattler. 2022. Adaptive Update Handling for Graph HTAP. In *ICDE Workshops 2022*. IEEE, 16–23. https://doi.org/10.1109/ICDEW55742.2022.00007

[41] Gary J. Katz and Joseph T. Kider Jr. 2008. All-Pairs Shortest-Paths for Large Graphs on the GPU. In *SIGGRAPH 2008*. EUROGRAPHICS/ACM, 47–55. https://doi.org/10.2312/EGGH/EGGH08/047-055

[42] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE 2011*. IEEE, 195–206. https://doi.org/10.1109/ICDE.2011.5767867

[43] James King, Thomas Gilray, Robert M. Kirby, and Matthew Might. 2016. Dynamic Sparse-Matrix Allocation on GPUs. In *ISC 2016*. Springer, 61–80. https://doi.org/10.1007/978-3-319-41321-1_4

[44] Tirthankar Lahiri, Shasank Chavan, Maria Colgan, Dinesh Das, Amit Ganesh, Mike Gleeson, Sanket Hase, Allison Holloway, Jesse Kamp, Teck-Hua Lee, Juan Loaiza, Neil MacNaughton, Vineet Marwah, Niloy Mukherjee, Atrayee Mullick, Sujatha Muthulingam, Vivekanandhan Raja, Marty Roth, Ekrem Soylemez, and Mohamed Zaït. 2015. Oracle Database In-Memory: A dual format in-memory database. In *ICDE 2015*. IEEE, 1253–1258. https://doi.org/10.1109/ICDE.2015.7113373

[45] Lina Lan, Ruisheng Shi, Bai Wang, Lei Zhang, and Jinqiao Shi. 2019. A Lightweight Time Series Main-Memory Database for IoT Real-Time Services. In *IOV 2019*. Springer, 220–236. https://doi.org/10.1007/978-3-030-38651-1_19

[46] Per-Åke Larson, Adrian Birka, Eric N. Hanson, Weiyun Huang, Michal Nowakiewicz, and Vassilis Papadimos. 2015. Real-Time Analytical Processing with SQL Server. *Proc. VLDB Endow.* 8, 12 (2015), 1740–1751. https://doi.org/10.14778/2824032.2824071

[47] Rubao Lee, Minghong Zhou, Chi Li, Shenggang Hu, Jianping Teng, Dongyang Li, and Xiaodong Zhang. 2021. The Art of Balance: A RateupDB Experience of Building a CPU/GPU Hybrid Database Product. *Proc. VLDB Endow.* 14, 12 (2021), 2999–3013. http://www.vldb.org/pvldb/vol14/p2999-lee.pdf

[48] Dean De Leo and Peter A. Boncz. 2021. Teseo and the Analysis of Structural Dynamic Graphs. *Proc. VLDB Endow.* 14, 6 (2021), 1053–1066. http://www.

vldb.org/pvldb/vol14/p1053-leo.pdf

[49] Chen Li, Rachata Ausavarungnirun, Christopher J. Rossbach, Youtao Zhang, Onur Mutlu, Yang Guo, and Jun Yang. 2019. A Framework for Memory Oversubscription Management in Graphics Processing Units. In *ASPLOS 2019*. ACM, 49–63. https://doi.org/10.1145/3297858.3304044

[50] Hang Liu and H. Howie Huang. 2019. SIMD-X: Programming and Processing of Graph Algorithms on GPUs. In *ATC 2019*. USENIX, 411–428. https://www.usenix.org/conference/atc19/presentation/liu-hang

[51] Zhidan Liu, Pengfei Zhou, Zhenjiang Li, and Mo Li. 2019. Think Like A Graph: Real-Time Traffic Estimation at City-Scale. *IEEE Trans. Mob. Comput.* 18, 10 (2019), 2446–2459. https://doi.org/10.1109/TMC.2018.2873642

[52] Darko Makreshanski, Jana Giceva, Claude Barthels, and Gustavo Alonso. 2017. BatchDB: Efficient Isolated Execution of Hybrid OLTP+OLAP Workloads for Interactive Applications. In *SIGMOD 2017*. ACM, 37–50. https://doi.org/10.1145/3035918.3035959

[53] Adam McLaughlin and David A. Bader. 2018. Accelerating GPU betweenness centrality. *Commun. ACM* 61, 8 (2018), 85–92. https://doi.org/10.1145/3230485

[54] Duane Merrill, Michael Garland, and Andrew S. Grimshaw. 2012. Scalable GPU graph traversal. In *PPOPP@SIGPLAN 2012*. ACM, 117–128. https://doi.org/10.1145/2145816.2145832

[55] Seungwon Min, Vikram Sharma Mailthody, Zaid Qureshi, Jinjun Xiong, Eiman Ebrahimi, and Wen-Mei Hwu. 2020. EMOGI: Efficient Memory-access for Out-of-memory Graph-traversal In GPUs. *Proc. VLDB Endow.* 14, 2 (2020), 114–127. https://doi.org/10.14778/3425879.3425883

[56] Aunn Raza, Periklis Chrysogelos, Panagiotis Sioulas, Vladimir Indjic, Angelos-Christos G. Anadiotis, and Anastasia Ailamaki. 2020. GPU-accelerated data management under the test of time. In *CIDR 2020*. http://cidrdb.org/cidr2020/papers/p18-raza-cidr20.pdf

[57] Viktor Rosenfeld, Sebastian Breß, and Volker Markl. 2022. Query Processing on Heterogeneous CPU/GPU Systems. *ACM Comput. Surv.* 55, 1, Article 11 (2022). https://doi.org/10.1145/3485126

[58] Arnon Rungsawang and Bundit Manaskasemsak. 2012. Fast PageRank Computation on a GPU Cluster. In *PDP 2012*. IEEE, 450–456. https://doi.org/10.1109/PDP.2012.78

[59] Amir Hossein Nodehi Sabet, Junqiao Qiu, and Zhijia Zhao. 2018. Tigr: Transforming Irregular Graphs for GPU-Friendly Graph Processing. In *ASPLOS 2018*. ACM, 622–636. https://doi.org/10.1145/3173162.3173180

[60] Mohammad Sadoghi, Souvik Bhattacherjee, Bishwaranjan Bhattacharjee, and Mustafa Canim. 2018. L-Store: A Real-time OLTP and OLAP System. In *EDBT 2018*. OpenProceedings.org, 540–551. https://doi.org/10.5441/002/edbt.2018.65

[61] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. 2020. The ubiquity of large graphs and surprising challenges of graph processing: extended survey. *VLDB J.* 29, 2-3 (2020), 595–618. https://doi.org/10.1007/s00778-019-00548-x

[62] Sherif Sakr, Angela Bonifati, Hannes Voigt, Alexandru Iosup, Khaled Ammar, Renzo Angles, Walid G. Aref, Marcelo Arenas, Maciej Besta, Peter A. Boncz, Khuzaima Daudjee, Emanuele Della Valle, Stefania Dumbrava, Olaf Hartig, Bernhard Haslhofer, Tim Hegeman, Jan Hidders, Katja Hose, Adriana Iamnitchi, Vasiliki Kalavri, Hugo Kapp, Wim Martens, M. Tamer Özsu, Eric Peukert, Stefan Plantikow, Mohamed Ragab, Matei Ripeanu, Semih Salihoglu, Christian Schulz, Petra Selmer, Juan F. Sequeda, Joshua Shinavier, Gábor Szárnyas, Riccardo Tommasini, Antonino Tumeo, Alexandru Uta, Ana Lucia Varbanescu, Hsiang-Yun Wu, Nikolay Yakovets, Da Yan, and Eiko Yoneki. 2021. The future is big graphs: a community view on graph processing systems. *Commun. ACM* 64, 9 (2021), 62–71. https://doi.org/10.1145/3434642

[63] Dipanjan Sengupta, Shuaiwen Leon Song, Kapil Agarwal, and Karsten Schwan. 2015. GraphReduce: processing large-scale graphs on accelerator-based systems. In *SC 2015*. ACM, 28:1–28:12. https://doi.org/10.1145/2807591.2807655

[64] Mo Sha, Yuchen Li, Bingsheng He, and Kian-Lee Tan. 2017. Accelerating Dynamic Graph Analytics on GPUs. *Proc. VLDB Endow.* 11, 1 (2017), 107–120. https://doi.org/10.14778/3151113.3151122

[65] Bin Shao, Haixun Wang, and Yatao Li. 2013. Trinity: a distributed graph engine on a memory cloud. In *SIGMOD 2013*. ACM, 505–516. https://doi.org/10.1145/2463676.2467799

[66] 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), 1281–1292. https://doi.org/10.14778/3007263.3007267

[67] Xuanhua Shi, Zhigao Zheng, Yongluan Zhou, Hai Jin, Ligang He, Bo Liu, and Qiang-Sheng Hua. 2018. Graph Processing on GPUs: A Survey. *ACM Comput. Surv.* 50, 6 (2018), 81:1–81:35. https://doi.org/10.1145/3128571

[68] Kshitij Shukla, Sai Charan Regunta, Sai Harsh Tondomker, and Kishore Kothapalli. 2020. Efficient parallel algorithms for betweenness- and closeness-centrality in dynamic graphs. In *ICS 2020*. ACM, 10:1–10:12. https://doi.org/10.1145/3392717.3392743

[69] Vishal Sikka, Franz Färber, Wolfgang Lehner, Sang Kyun Cha, Thomas Peh, and Christof Bornhövd. 2012. Efficient transaction processing in SAP HANA database: the end of a column store myth. In *SIGMOD 2012*. ACM, 731–742. https://doi.org/10.1145/2213836.2213946

[70] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stanley B. Zdonik. 2005. C-Store: A Column-oriented DBMS. In *VLDB 2005*. ACM, 553–564. http://www.vldb.org/archives/website/2005/program/paper/thu/p553-stonebraker.pdf

[71] Gábor Szárnyas, Arnau Prat-Pérez, Alex Averbuch, József Marton, Marcus Paradies, Moritz Kaufmann, Orri Erling, Peter A. Boncz, Vlad Haprian, and János Benjamin Antal. 2018. An early look at the LDBC social network benchmark's business intelligence workload. In *GRADES@SIGMOD 2018*. ACM, 9:1–9:11. https://doi.org/10.1145/3210259.3210268

[72] Vibhav Vineet and P. J. Narayanan. 2008. CUDA cuts: Fast graph cuts on the GPU. In *CVPR 2008*. IEEE, 1–8. https://doi.org/10.1109/CVPRW.2008.4563095

[73] Hao Wang, Dogan Can, Abe Kazemzadeh, François Bar, and Shrikanth S. Narayanan. 2012. A System for Real-time Twitter Sentiment Analysis of 2012 U.S. Presidential Election Cycle. In *ACL System Demonstrations 2012*. 115–120. https://aclanthology.org/P12-3020/

[74] Hao Wang, Liang Geng, Rubao Lee, Kaixi Hou, Yanfeng Zhang, and Xiaodong Zhang. 2019. SEP-graph: finding shortest execution paths for graph processing under a hybrid framework on GPU. In *PPoPP@SIGPLAN 2019*. ACM, 38–52. https://doi.org/10.1145/3293883.3295733

[75] Kai Wang, Don Fussell, and Calvin Lin. 2021. A fast work-efficient SSSP algorithm for GPUs. In *PPoPP@SIGPLAN 2021*. ACM, 133–146. https://doi.org/10.1145/3437801.3441605

[76] Kaibo Wang, Kai Zhang, Yuan Yuan, Siyuan Ma, Rubao Lee, Xiaoning Ding, and Xiaodong Zhang. 2014. Concurrent Analytical Query Processing with GPUs. *Proc. VLDB Endow.* 7, 11 (2014), 1011–1022. https://doi.org/10.14778/2732967.2732976

[77] Pengyu Wang, Jing Wang, Chao Li, Jianzong Wang, Haojin Zhu, and Minyi Guo. 2021. Grus: Toward Unified-memory-efficient High-performance Graph Processing on GPU. *ACM Trans. Archit. Code Optim.* 18, 2 (2021), 22:1–22:25. https://doi.org/10.1145/3444844

[78] Yangzihao Wang, Andrew A. Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. 2016. Gunrock: a high-performance graph processing library on the GPU. In *PPoPP@SIGPLAN 2016*. ACM, 11:1–11:12. https://doi.org/10.1145/2851141.2851145

[79] Martin Winter, Daniel Mlakar, Rhaleb Zayer, Hans-Peter Seidel, and Markus Steinberger. 2018. faimGraph: high performance management of fully-dynamic graphs under tight memory constraints on the GPU. In *SC 2018*. IEEE / ACM, 60:1–60:13. http://dl.acm.org/citation.cfm?id=3291736

[80] Martin Winter, Rhaleb Zayer, and Markus Steinberger. 2017. Autonomous, independent management of dynamic graphs on GPUs. In *HPEC 2017*. IEEE, 1–7. https://doi.org/10.1109/HPEC.2017.8091058

[81] Tianji Wu, Bo Wang, Yi Shan, Feng Yan, Yu Wang, and Ningyi Xu. 2010. Efficient PageRank and SpMV Computation on AMD GPUs. In *ICPP 2010*. IEEE, 81–89. https://doi.org/10.1109/ICPP.2010.17

[82] Chang Ye, Yuchen Li, Bingsheng He, Zhao Li, and Jianling Sun. 2021. GPU-Accelerated Graph Label Propagation for Real-Time Fraud Detection. In *SIGMOD 2021*. ACM, 2348–2356. https://doi.org/10.1145/3448016.3452774

[83] Kangfei Zhao and Jeffrey Xu Yu. 2017. All-in-One: Graph Processing in RDBMSs Revisited. In *SIGMOD 2017*. ACM, 1165–1180. https://doi.org/10.1145/3035918.3035943

[84] Xiaowei Zhu, Marco Serafini, Xiaosong Ma, Ashraf Aboulnaga, Wenguang Chen, and Guanyu Feng. 2020. LiveGraph: A Transactional Graph Storage System with Purely Sequential Adjacency List Scans. *Proc. VLDB Endow.* 13, 7 (2020), 1020–1034. https://doi.org/10.14778/3384345.3384351