Bridging the Gap between Relational OLTP and Graph-based OLAP

Sijie Shen[1,2], Zihang Yao[1], Lin Shi[1], Lei Wang[2], Longbin Lai[2], Qian Tao[2], Li Su[2], Rong Chen[1,3], Wenyuan Yu[2], Haibo Chen[1], Binyu Zang[1], and Jingren Zhou[2]

[1]Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
[2]Alibaba Group
[3]Shanghai AI Laboratory

Abstract

Recently, many applications have required the ability to perform dynamic graph analytical processing (GAP) tasks on the datasets generated by relational OLTP in real time. To meet the two key requirements of performance and freshness, this paper presents GART, an in-memory system that extends hybrid transactional/analytical processing (HTAP) systems for hybrid transactional and graph analytical processing (HTGAP). GART fulfills two unique goals that are not encountered by HTAP systems. First, to adapt to rich workloads flexibility, GART proposes transparent data model conversion by graph extraction interfaces, which define rules for relational-graph mapping. Second, to ensure GAP performance, GART proposes an efficient dynamic graph storage with good locality that stems from key insights into HTGAP workloads, including (1) an efficient and mutable compressed sparse row (CSR) representation to guarantee the locality of edge scan, (2) a coarse-grained multi-version concurrency control (MVCC) scheme to reduce the temporal and spatial overhead of versioning, and (3) a flexible property storage to efficiently run different GAP workloads. Evaluations show that GART performs several orders of magnitude better than existing solutions in terms of freshness or performance. Meanwhile, for GAP workloads on the LDBC SNB dataset, GART outperforms the state-of-the-art general-purpose dynamic graph storage (i.e., LiveGraph) by up to 4.4×.

1 Introduction

Graphs, due to their natural ability to model intricate relations among entities [12, 61], have been intensively adopted to model business data. Correspondingly, *graph analytical processing (GAP)* techniques are being developed to better understand graph data and are widely applied in many fields, such as recommendation systems [70, 74], supply-chain analysis [46], and fraud detection [29, 56]. As business data is constantly generated and updated, there calls for an urgent need for *dynamic* GAP workloads on *real-time* datasets. In many traditional business scenarios, data is usually updated by online transaction processing (OLTP) in relational databases [26, 47, 82].

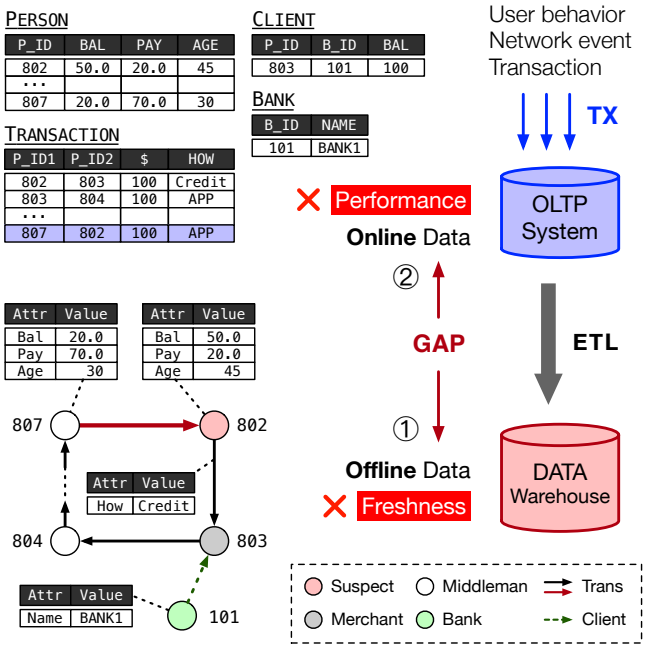

Fig. 1. *A comparison of solutions for dynamic graph analytical processing* (GAP) *on transactional datasets and their limitations.* **Solutions***: graph processing on offline data* (①) *or online data* (②).

Real-world Example. In Fig. 1, we demonstrate a simplified online credit card fraud detection task in e-commerce platforms [56], in which a suspect attempts to obtain short-term credit from a credit card via illegal transactions. To achieve this, the suspect (802) makes sham purchases paid by a credit card from a conspired merchant (803). The merchant, after receiving the money from the bank (101), transfers the money through a series of middlemen (804, . . . , 807) with other fraudulent transactions back to the suspect (802). In this scenario, the OLTP system maintains four tables (PERSON, TRANSACTION, CLIENT, and BANK), from which one can create a graph showing relationships of transactions among normal users, suspects, merchants, and middlemen. Whenever a new tuple occurs in the TRANSACTION table, the graph should be updated correspondingly. As soon as an upcoming transaction generates a cycle in the graph, an alarm should be triggered instantly to block the transaction for fur-

ther investigation. Therefore, an underlying detection system for such frauds should meet two key requirements simultaneously.

Performance. The *performance degradation* of both relational transaction and graph analytical workloads should be minimal compared to running them separately on specific systems. Thereby, both the transaction and detection should be completed before the user perceives any lag.

Freshness. The *time gap* between transactions committed on OLTP systems and their accessibility on detection systems should be minimal to prevent fraud on time. Recent studies [10, 56] show extreme requirements of 20-millisecond freshness for fraud detection or system monitoring.

In response, several solutions have been proposed to support such workloads. Unfortunately, none of them can simultaneously meet the requirements, as shown in Fig. 1.

Solution ①: graph processing on offline data. To achieve better GAP performance, this solution utilizes existing graph-specific systems (particularly for *static* graphs in many cases) [22, 31, 67, 76, 86], such as GraphScope [29], to handle GAP workloads efficiently. Since data is separately maintained in OLTP and graph-specific systems, an offline data migration with an ETL (Extract-Transform-Load) process is required. However, such a process is often expensive and slow, and results in a high lag between the transactional data in OLTP systems and the extracted graph data in graph-specific systems [82], which deteriorates the *freshness* guarantee.

Solution ②: graph processing on online data. Some OLTP systems [34, 37, 55, 84] attempt to translate graph-related operations into relational operations. However, prior work [21, 73] has found this solution causes *performance* degradation of GAP up to several orders of magnitude due to costly join operations and huge redundant intermediate data. On the other hand, graph databases [6, 7, 27] use a native graph representation to ensure the efficiency of GAP workloads and directly commit transactions on graphs. However, due to the more complicated management (e.g., maintaining adjacency lists instead of inserting a row) in graph databases to fulfill transactions [87], the *performance* of transactions in graph databases is significantly slower than the relational counterparts, which is also demonstrated in our experimental study (§6.2). Further, legacy business logic was usually designed and implemented on relational OLTP systems, and it is inevitable to process a costly migration to the graph databases.

The tradeoff between performance and freshness is still an open problem for dynamic GAP workloads. Fortunately, *hybrid transactional/analytical processing* (HTAP) is a new trend that processes OLTP and online analytical processing (OLAP) simultaneously in the same system. The state-of-the-art HTAP systems usually leverage a loosely-coupled design to guarantee both performance and freshness [19, 38, 45, 50, 65, 82], which gives an opportunity for dynamic GAP workloads. Analogously, we term dynamic GAP workloads on transactional datasets as *hybrid transactional/graph-analytical processing* (HTGAP).

Our approach. This paper presents GART, an in-memory HTGAP system extended from HTAP systems that can be deployed to bridge an existing relational OLTP system with a graph-specific system for requirements of *performance* and *freshness*. GART performs GAP workloads over the graph-specific system with little *performance* degradation. It reuses transaction logs to replay graph data *online* for *freshness* instead of offline data migration. Unlike the prior HTAP systems with only the relational model, GART also has to support the graph data model for GAP. Therefore, there are two unique goals not encountered by HTAP systems.

First, to adapt to rich workloads flexibly, GART needs to convert relational data to graph data transparently. Thus, some concise yet expressive interfaces should be proposed to the database administrator (DBA) for data conversion from the relational model to the graph model. To fulfill this goal, we propose a collection of *graph extraction interfaces* to define rules of relational-graph mapping in a newly designed component called *RGMapping*. We demonstrate that the interfaces are expressive enough to help GART automatically extract property graphs from relational data sources such as transactions.

Second, to guarantee performance, the dynamic graph storage should support both read and write operations efficiently. Existing *general-purpose* dynamic graph storages [26, 33, 87] support complex updates from transactions but provide sub-optimal GAP performance due to poor data locality and expensive concurrency control. Thus, based on observed characteristics of HTGAP workloads, we propose a new efficient dynamic graph storage for HTGAP with three key components: 1) an *efficient* and *mutable* CSR representation that guarantees the locality of edge scan when updating graph topology data; 2) a *coarse-grained* MVCC scheme that reduces the temporal and spatial overhead of versioning; 3) a *flexible* property storage that allows running different GAP workloads on snapshots generated based on their access patterns.

We implement GART by extending VEGITO [65], a state-of-the-art in-memory HTAP system. The extensions include graph extraction interfaces, the dynamic property graph storage, and integrating a unified graph computation system (GraphScope [29]). To demonstrate the efficacy of GART, we have conducted a set of experiments using two popular benchmarks (i.e., LDBC SNB [4] and TPC-C [71]), as well as diverse graph datasets. Our experimental results show that GART outperforms Solution ① and Solution ② by several orders of magnitude in freshness and performance, respectively. Meanwhile, GART outperforms the state-of-the-art general-purpose dynamic graph storage (i.e., LiveGraph [87]) by up to 4.4× for GAP workloads on the LDBC SNB dataset.

Contributions. We extend HTAP systems to support

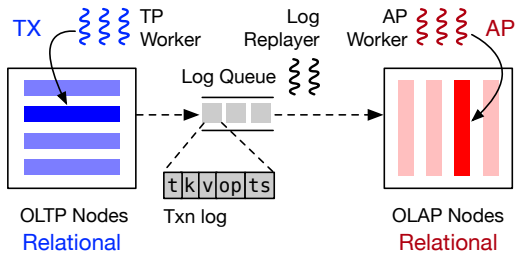

Fig. 2. *The overview of HTAP systems based on a loosely-coupled design.* ***Transaction (Txn) log****: table ID (t), primary key (k), value (v), operation type (op), and timestamp (ts).*

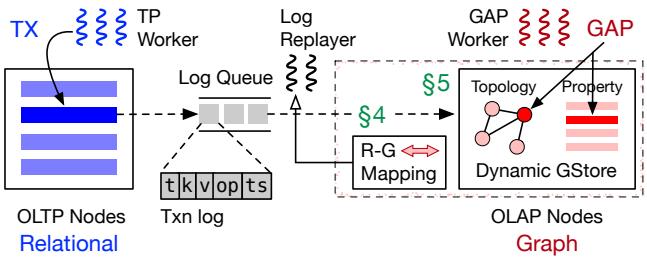

Fig. 3. *The architecture of* GART*. The components in the dashed box are newly designed for HTGAP.*

HTGAP by incorporating relational-graph mapping and a dynamic graph storage into existing computation engines. In summary, the contributions of this paper are:

- The first to extend HTAP architecture for HTGAP workloads with guarantees of performance and freshness (§3) that proposes expressive interfaces of relational-graph mapping for transparent data model conversion (§4).
- A new dynamic graph storage for efficient HTGAP workloads, which is optimized for data locality and concurrency control based on our key insights into HTGAP (§5).
- A prototype implementation (GART) that integrates existing HTAP and GAP systems, as well as a set of evaluations that confirm the efficacy of GART for HTGAP workloads with diverse applications and datasets (§6).

2 Opportunity: HTAP

Hybrid transactional/analytical processing (HTAP) is a new trend that bridges the gap between OLTP and OLAP for real-time analytics on datasets updated by transactions and is already being used in many scenarios [17, 53, 85].

The *loosely-coupled* design is a common choice of state-of-the-art HTAP systems, which dedicate OLTP and OLAP to different physical resources with specific storage types (see Fig. 2). Thereby, it is comparable in *performance* to specific execution engines and can synchronize data with transaction logs to guarantee *freshness* [38, 45, 50, 65, 82]. Specifically, logs of the OLTP node are used to update the extra *column* store on the backup (OLAP node), which is more appropriate for OLAP workloads; the log replayer applies logs in real time on the OLAP node. The log from the OLTP system (*Txn log* in Fig. 2) contains the necessary information for data replaying, such as the identifier of data updates (table ID and primary key), the after-image or delta of the tuple (value), and the temporal meta-data (version number or timestamp) [23, 52, 63, 66, 78]. Reusing logs for HTAP can avoid costly operations for change data capture (CDC).

HTAP systems usually utilize batch-based log replaying for freshness and consistency on the OLAP storage [45, 50, 65]. An efficient design [65] divides time into consecutive and non-overlapping *epochs*. The epoch is automatically increased in a fixed interval, such as several milliseconds, that can trade off between freshness and performance. During each epoch, logs can be replayed in parallel. When an epoch ends, it guarantees all the logs within the epoch have been replayed. Each epoch is specified by an epoch number, which is incremented when entering a new epoch. The system maintains a write epoch number (i.e., wepoch) to represent the epoch when the logs are being replayed. The latest stable epoch number that OLAP users can read is latest_repoch.

Opportunity. We observe that the guarantees of HTAP systems on both performance and freshness can benefit the dynamic GAP workloads. HTAP systems can be extended to process transactions and GAP workloads simultaneously on different storages. Specifically, for **performance**, HTAP systems provide specific storages and execution engines for hybrid workloads, which can fully reuse the effort on specific systems for OLTP and GAP. For **freshness**, logs in HTAP systems contain the updates of relational data, which can be synchronously applied to the graph data in real time without costly operations such as bulk loading, CDC and ETL.

However, to the best of our knowledge, none of the HTAP systems enable native GAP workloads on a dynamic graph storage. To achieve this, HTAP systems need to support conversion between the relational model and graph data model and an efficient dynamic graph storage for HTGAP.

3 Overview of GART

Inspired by the loosely-coupled design of HTAP systems [38, 65, 82], we propose GART, an in-memory HTGAP system that extends HTAP systems by retrofitting the log replayer and the storage for GAP workloads, as shown in Fig. 3. It should be noted that GART can reuse the execution engines of existing OLTP and graph-specific systems.

Architecture and workflow. In GART, transactions are committed in the OLTP nodes and generate logs, like prior HTAP systems [38, 65, 82]. To support rich workloads flexibly and efficiently, GART conducts data model conversion. Relational data in logs need to be converted to graph data and stored in a dynamic graph storage (GStore) of OLAP nodes. GART allows the DBA, who is responsible for defining the database schema, to define the relational-graph mapping through the *RGMapping* component. RGMapping guides the *log replayer* to convert the relational data in logs to the updates on graph data.

GART devises a new *dynamic graph storage* for real-time graph updates and different GAP workloads. Graph data consists of a topology and properties. The topology contains vertices and edges (i.e., an ordered pair of vertices), and the properties are a set of attributes for each vertex or edge. The storage always provides consistent snapshots of graph data (identified by an *epoch*) derived from relational data. Similar to HTAP systems (§2), the log replayer updates the storage with the epoch number wepoch. GAP workloads can read a fresh snapshot or earlier using an epoch number that does not exceed latest_repoch.

GART follows a loosely-coupled design, which can be deployed as a single-machine system or a distributed system that separates OLTP and GAP components (the dashed box in Fig. 3) on different machines for better performance isolation. For the distributed deployment, a crash of the GAP component will not stall the execution of the OLTP component. The graph data can be recovered from the relational data according to persistent RGMapping data. When the OLTP component fails, the existing fault-tolerance mechanism in HTAP systems still works [65], which is orthogonal to our work. In addition, we focus on in-memory processing that can buffer hot data in real-time GAP tasks and meet the freshness and performance requirements. GART is independent of whether the OLTP system is in-memory or not.

To support HTGAP workloads, GART should fulfill two unique design goals never encountered in prior work.

Goal 1: Transparent data model conversion (§4). In HTAP systems, the conversion does not change the data model and only depends on the schema of relational data (e.g., from row store to column store [38, 65]). However, the conversion between different data models for HTGAP workloads requires more semantic information. For example, it needs the mapping between relational tables and vertex/edge types, and the mapping between relational attributes and vertex/edge properties. Prior work [37, 55, 72] uses interface extension rather than data conversion, such as graph extensions on relational databases, which demands users to manually rewrite transactions or change log formats.

Goal 2: Efficient dynamic graph storage (§5). For the HTGAP system, write operations (from the log replayer) and read operations (from the GAP worker) are executed concurrently on the graph storage. The performance of both is important. Although many general-purpose dynamic graph storage systems [30, 32, 54, 87] have existed, their read performance for GAP workloads is sub-optimal due to neglect of HTGAP characteristics. The locality of read operations is sacrificed to guarantee the write performance and transaction semantics. First, adjacency-list-based topology storages ignore the locality of edge scan. Second, fine-grained versioning is expensive and breaks both the spatial and temporal locality. Third, property storages based on a column store cannot guarantee the locality of access patterns among different GAP workloads.

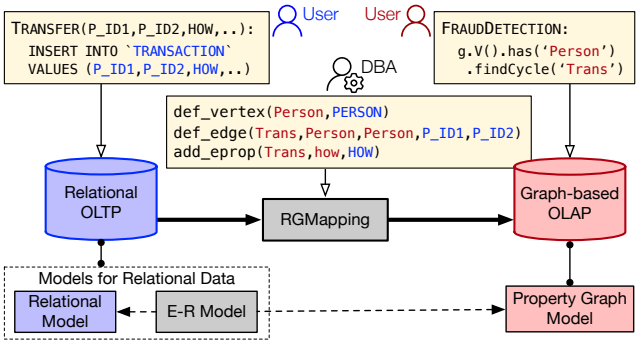

Fig. 4. *An example of data manipulation and graph extraction interfaces provided by* GART *using the dataset in Fig. 1.*

4 RELATIONAL-GRAPH MAPPING

To convert relational data to graph data automatically, it is necessary to provide a relational-graph mapping mechanism. GART uses the *property graph model* and provides the interfaces with the intuition from the *entity-relationship (E-R) model*. The property graph model [11] has been widely adopted to model graph-structured data. As shown in the lower left corner of Fig. 1, a property graph model defines a directed graph topology in which a vertex represents an entity and an edge from a source vertex to a target vertex represents a relationship. Each vertex (resp. edge) belongs to a vertex (resp. edge) type and has a property with attributes (Attr-Value pairs).

4.1 System Interfaces

The conversion from relational data to graph data needs additional semantic information. An intuitive solution is to directly add graph information to the transactions, such as graph extensions for relational databases [48, 55, 69], so that additional information can be added to the log. However, this solution has to extend the interface of the OLTP engine and change the log format; it implies that transactions must also be rewritten manually. Instead, GART decouples the interface into two groups, which are exposed to the user and the DBA, as shown in Fig. 4.

Data manipulation interfaces. GART integrates specific execution engines for OLTP and GAP workloads and retains their user interfaces. Therefore, existing transactions and graph queries can run directly on GART. As the example in Fig. 4 shows, users can execute a transaction called TRANSFER to transfer money. Meanwhile, users can run a query called FRAUDDETECTION to find all cycles on the graph consisting of Person vertices and Trans edges.

Graph extraction interfaces. The interfaces of the *RGMapping* component define the relational-graph mapping, which guides the log replayer to perform data conversion automatically. Fig. 5 lists two kinds of graph extraction interfaces.

Interfaces for adding vertices. In GART, each vertex type corresponds to one table in the relational model, and each

```
# Definition for vertices
def_vertex(vtype,table)
add_vprop(vtype,vprop,attr)
# Definition for edges
def_edge(etype,src_vtype,dst_vtype,pk) # 1-to-m
def_edge(etype,src_vtype,dst_vtype,src_pk,dst_pk) # m-to-m
add_eprop(etype,eprop,attr)
```

Fig. 5. *The graph extraction interfaces provided by* GART. *Arguments from the graph model and the relational model are shown in red and blue, respectively.*

property of vertices corresponds to one attribute in the table. The interfaces def_vertex and def_vprop are used to add new entities and construct the corresponding vertices. Specifically, def_vertex defines a type of vertices (vtype) according to the corresponding table (table). The attributes (attr) of the table can be further mapped to the properties (vprop) of vertices through the interface add_vprop.

Interfaces for adding edges. A relationship in the relational model can be added as a directed edge through the interface def_edge, where etype, src_vtype and dst_vtype correspond to the edge type, the type of source and destination vertices of this type of edges, respectively. To distinguish between different relationship types, RGMapping provides two def_edge for 1-to-m relationships (also 1-to-1 relationships) and m-to-m relationships, respectively. The difference between them lies in whether the interface requires the primary keys (pk) of one table or both tables as inputs. Furthermore, add_eprop is used to add the edge property (eprop).

The graph extraction interfaces make the OLTP engine and log formats unchanged. Moreover, unlike data manipulation interfaces, graph extraction interfaces are used only when defining the graph schema instead of used in each request. DBAs can define the RGMapping for a fixed data model just once according to workloads. For complex data models, DBAs can use automatic E-R model generation tools [1, 5] as a guide according to the relational schema, even if they have less knowledge about the workloads.

4.2 Expressiveness of RGMapping

We next show that graph extraction interfaces are expressive enough to map relational data to a property graph modeled by the same E-R model.

The E-R model has shown its powerful expressive capability in describing relationships and has been widely adopted in defining relational data [20, 28]. Generally speaking, the E-R model contains a set of *entities* with *attributes*, which usually represent objects in the physical world, and describes the *relationships* between entities. Intuitively, *for any E-R model, there exists a unique property graph schema that represents the E-R model* [59]. Entities and relationships can be mapped to vertex types and edge types of the property graph model, respectively, and use properties to store attributes. The interfaces also help DBAs extract a subgraph from the property graph. Note that edges in graphs are binary (2-ary) relations. An n-ary relationship of order greater than two can

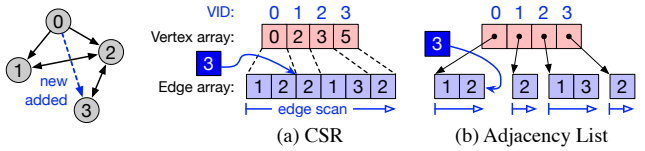

Fig. 6. *Two typical representations of an example dynamic graph topology, namely (a) CSR and (b) adjacency list.*

be mapped as a type of vertices with n associated edges.

Back to the example in Fig. 1 and Fig. 4, assume that a mapping scheme between the E-R model and the relational model has been defined (lower left dashed rectangles in Fig. 4). There is an entity called Person and a relationship called Trans that exists between instances of the Person entity in the E-R model. The DBA can utilize the interfaces (the middle part in Fig. 4) to define a property graph model that contains one type of vertices (Person) and one type of edges (Trans), which are derived by the entity Person and relationship Trans, respectively. Meanwhile, there is a property (How) on Trans derived from the HOW attribute.

RGMapping can map changes to relational data to property graphs *on-the-fly*. Depending on whether the data involves entity tables or relationship tables, the log replayer converts them to vertices or edges, respectively. Users can customize the extracted graphs partially so that the graphs extracted do not have to exactly match the E-R model.

5 Dynamic Graph Storage

The graph storage of GART stores the graph topology and properties and provides two kinds of operations: read for GAP workloads (e.g., edge scan) and write for the log replayer (e.g., insert and delete). For the graph topology, compressed sparse row (CSR), a compact graph representation, is widely adopted by (static) graph systems [29, 43, 73, 77], as shown in Fig. 6(a). However, CSR is also notoriously inefficient for dynamic workloads on the graph (e.g., edge insertion and deletions). Therefore, dynamic graph storage systems [30, 32, 33, 41, 54, 87] commonly use adjacency lists (based on linked lists or vectors) to store the graph topology (see Fig. 6(b)). For vertex (resp. edge) properties, a columnar storage is usually employed to efficiently read the same property for all vertices (resp. edges) with the same type [29, 55]. In addition, the dynamic graph storage also needs to record the version of vertex/edge/property updates for MVCC [30, 87].

The above traditional design has several performance issues for HTGAP workloads. First, using adjacency lists suffers from poor locality when sequentially scanning edges of all vertices, which is a common yet costly operation in GAP. A cache miss may occur when scanning the edges of an adjacent vertex. For example, transactions may insert edges randomly, resulting in unordered memory allocation for new edges of different vertices. Second, using fine-grained versioning for each vertex or edge update imposes a significant performance penalty for GAP workloads, since checking the

version for each read operation breaks both spatial and temporal locality and introduces additional overhead. Third, the existing property storages cannot guarantee locality flexibly for different access patterns among GAP workloads.

Key insights. Some unique characteristics of HTGAP workloads open opportunities to exploit the locality of a dynamic graph storage. Note that we term the time gap as the interval between a transaction committing an update and a graph query reading it. First, *the time gap in HTGAP (typically a dozen milliseconds, which is equal to the freshness of* GART *shown in Table 2) is sufficient to update a compact structure like CSR*. Thus, GART can still use a CSR-like storage for the graph topology instead of adjacency lists to improve the locality of edge scan. Second, *the GAP latency is almost always much longer than the time gap*. It implies that assigning versions to each update (fine-grained MVCC) is not necessary for GAP workloads. Thus, GART can use coarse-grained MVCC (i.e., at epoch granularity) to reduce memory and computation overhead, even though the committed updates cannot be read immediately. Third, *the access pattern of each GAP workload is usually fixed and easily detectable*. Given an HTGAP workload, fixed correlations between different attributes can be found by parsing the requests. For example, some attributes (e.g., balance and payment) may always be updated or read together. Therefore, GART can allow users to decide how to store different properties.

General idea. Based on the insights, we devise a new dynamic graph storage for HTGAP workloads. Fig. 7 illustrates the main structure of the dynamic graph storage for one type of vertex and edge. For the graph topology, a variant of CSR is proposed to exploit locality of edge scan, where the edge array is divided into multiple *edge segments* for dynamic updates. Using edge segments offers a tradeoff between read and write performance and allows the structure to be updated in batches. The vertex array is indexed by vertex ID (VID) and contains the links to the edges (neighbors) of each vertex. To reduce the overhead of edge insertion, the edges of each vertex are further divided into multiple *edge blocks*. Each vertex stores a pointer (tail) to the last edge block in the vertex array (not shown in Fig. 7 due to space constraints). To enable MVCC at epoch granularity, each vertex maintains an *epoch table*, which links to the edges inserted in the same epoch. It avoids attaching versions to each edge as in fine-grained MVCC. Similar to CSR, the epoch table stores the logical offset of the first edge for each (read) epoch number. In addition, vertex (resp. edge) properties are stored in *property blocks* within the vertex array (resp. edge segment). Each property block is a column store for one property or a group of correlative properties (column-family), which is indexed by the corresponding vertex (resp. edge) offset in the vertex array (resp. edge segment). Finally, a new interface is provided for combining correlative properties into a column family following access patterns of HTGAP workloads.

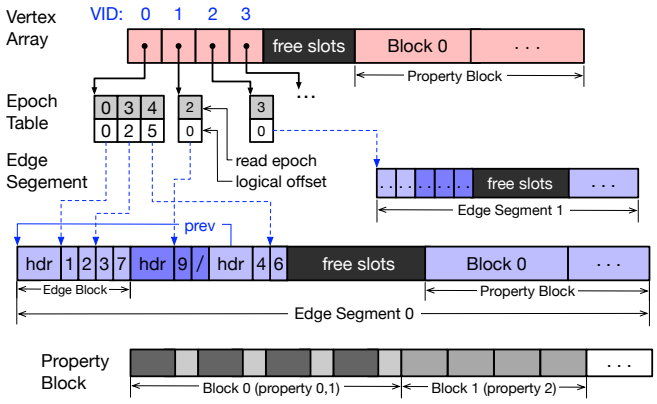

Fig. 7. *The key structure of the dynamic graph storage in* GART *for one type of vertex and edge.*

5.1 Efficient and Mutable CSR

GART devises an efficient and mutable CSR that guarantees high performance in both scans and updates on the graph topology, providing data locality similar to an immutable CSR. Each edge segment has a fixed initial size (e.g., 16KB) and stores edges (i.e., neighboring vertex IDs) of a fixed number of vertices (e.g., 4,096). The free slots are reserved for new edge blocks. Each vertex has a group of *edge blocks*, and new edges will be inserted into the tail edge block.

Fig. 7 shows an example where each segment stores the edges of two vertices (e.g., vertex 0 and 1). Initially, edges of the same vertex are stored consecutively in an edge block, so there is no overlap between edges of different vertices in an edge segment. As edges are continuously inserted, a vertex will allocate new edge blocks, which form a linked list (e.g., vertex 0). Each edge block has a header block (hdr) to store the meta-data of edges, such as the block size, the number of valid edges, and the pointer (prev) to the previous edge block of the same vertex.

Edge scan. Given a read epoch number, GART first uses the tail pointer and the epoch table of the vertex to find edges of that epoch within its edge blocks, and then scans edges forward based on the prev pointer stored in the header block (hdr). Note that each edge block only needs to be addressed once, which has little performance impact. In the beginning, GART can provide data locality comparable to vanilla CSR. However, after inserting numerous edges for different vertices, the edge blocks of a vertex will form a long linked list, leading to performance degradation in edge scan. To mitigate this issue, GART compacts an edge segment periodically or when the segment is close to full. After compaction, all edge blocks of the same vertex will be merged into one edge block (e.g., the first edge block of vertex 0).

Insertion. For a new vertex, GART atomically inserts it into a free slot of the vertex array and initializes an empty epoch table for it. When inserting an edge (e.g., from vertex 1 to vertex 4), the destination vertex ID will be directly inserted

into the tail edge block of the source vertex, if the edge block is not full (e.g., vertex 1 in Fig. 7). Otherwise, a new tail edge block of double the size is first allocated from the free slot of the edge segment, and then the edge is inserted into it. To reduce fragmentation, when the size of the tail edge block is smaller than a threshold, all edges will be moved to the newly allocated edge block to further improve data locality; the original edge block will be skipped. When an edge segment is full, a new segment of double the size is allocated, and all edges in the original segment are moved to the new one.

The write conflicts when concurrently inserting edges to the same vertex are resolved by per-vertex locks. Moreover, to resolve the conflicts when allocating edge blocks for different vertices on a full segment, the log replayer should lock the segment after checking for free space. Specifically, if the segment still has free space, the segment is locked in a shared manner; if the segment is full, the log replayer should exclusively lock the segment first and then allocates a new one.

Deletion. When deleting a vertex (resp. edge), a delete flag is appended to the vertex array (resp. edge block), which records the offset of the deleted vertex (resp. edge). When encountering the delete flag, the deleted vertex (resp. edge) will be skipped during scanning the graph topology. Furthermore, garbage collection (GC) will physically delete the vertices and edges and free up space in the background.

Discussion: structure parameters. We have tuned parameters including: (1) the number of vertices managed by each segment, (2) the initial size of edge blocks and segments, and (3) the resize factor of edge blocks (or segments) when they are full. Increasing (1) results in higher latencies for inserts, but it improves read performance. To balance read and write performance, we set (1) to 4096. The default values of (2) and (3) have minimal impact on read performance. We have adjusted these values to minimize the allocation of edge segments and optimize memory usage.

5.2 Coarse-grained MVCC

GART employs a *coarse-grained* MVCC scheme to reduce the temporal and spatial overhead of fine-grained MVCC. The scheme is based on the key observation that GAP workloads usually run longer at low concurrency than transactions. Thus, GART can enable MVCC at epoch granularity. In particular, the edge storage needs to adapt to the epoch instead of a fine-grained version for each edge. For each vertex, the epoch number of edges increases with the logical offsets. Since edges are append-only in edge blocks, edges with the same epoch number are consecutive. Therefore, GART can use an epoch number for a batch of edges.

Specifically, each vertex maintains an *epoch table* to store the offset in the edge segment for each epoch. As the example shown in Fig. 7, the 3rd to 5th edges of vertex 0 (offsets 2–4) are all inserted at epoch 3. At epoch 4, the GAP worker

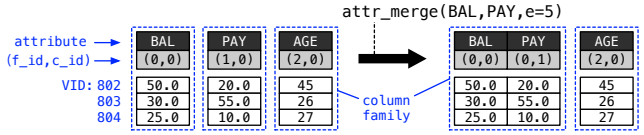

Fig. 8. *An example of the flexible property storage.*

will read all edges with the logical offset less than 5 at vertex 0. Since the logical offset of each epoch is immutable, GART can scan edges sequentially as if on a static graph. This design avoids checking versions for each edge and maintains the compact edge storage like in CSR.

When the log replayer inserts an edge with the epoch number, it will check whether the epoch number exists in the epoch table of the source vertex. The transaction protocol guarantees that the new epoch number must be greater than all existing epoch numbers [65]. If the epoch number does not exist, a new epoch number and the offset will be appended to the epoch table atomically.

The epoch table is stored as a ring buffer, since older snapshots are more likely to never be read again. For the corner cases where the oldest epoch number is still in use, GART uses a classical allocation amortization technique, similar to the C++ STL vector, to extend the ring buffer.

5.3 Flexible Property Storage

The storage model of properties influences the performance of read operations over properties. However, there exists no efficient property storage model for all GAP workloads. To support different GAP workloads in a more efficient way, GART presents a *flexible property storage* that allows system users to define the storage model according to application memory access patterns. Initially, GART utilizes a column store to store property data, which stores the same property (attribute) of the same type of vertices/edges continuously and is friendly to workloads that scan one or several attributes sequentially and independently.

To improve the property scan performance under different scenarios, users can combine several high-related attributes into a *column family* with the attr_merge interface *ahead of time*, as shown in Fig. 8. The Person vertex consists of three attributes: Bal, Pay, and Age. The meta-data of each attribute maintains the column family index (f_id) and the column index in the column family (c_id). The initial f_ids are different due to the pure column store.

As time goes on, the access patterns to some attributes may have certain correlations. Users can use the attr_merge interface to merge some attributes into the same column family *on-the-fly*, and generate a snapshot with a read epoch number. For example, if the attributes Bal and Pay are accessed together in a long-term GAP workload, users can merge them into a column family with read epoch 5. GART needs to generate a new version of the meta-data of these two attributes with epoch number 5, and copy them into a column family in the background. Then, the workload is processed on the new

Table 1: *Datasets used in evaluation, including TPC-C (warehouse=20) [71]* (CH)*, SNB-SF-10 [4]* (SB)*, Wiki [36]* (WK)*, R-MAT [18]* (RM)*, UK-2005 [16]* (UK)*, and Twitter-2010 [42]* (TT)*.*

Graphs	CH	SB	WK	RM	UK	TT
$\|V\|$	700 K	7.5 M	5.7 M	5.0 M	39.5 M	41.7 M
$\|E\|$	6.0 M	8.8 M	130 M	300 M	936 M	1.47 B

snapshot, which can also avoid some pre-processing tasks (e.g., the projection phase of GNN workloads to obtain required properties). We leave the automatic plan generation and attribute merging as future work.

6 Evaluation

We have implemented GART by extending VEGITO [65], a state-of-the-art in-memory HTAP system. VEGITO adopts DrTM+H [79] as the OLTP engine and supports tens-of-millisecond freshness with millions of transactions per second. This makes the HTGAP system design more challenging than in the case of low OLTP throughput. The extensions include the log replayer with relational-graph mapping and a new dynamic graph storage. We further integrated an open-sourced one-stop graph processing engine GraphScope [29] into GART, in order to support diverse GAP workloads.

6.1 Experimental Setup

Testbed. All evaluations are conducted on two dual-socket machines. Each machine has two 12-core Intel Xeon E5-2650 CPUs, 256 GB DRAM, and two 56 Gbps InfiniBand (IB) NICs via PCIe 3.0 connected to a Mellanox 40 Gbps IB Switch. Since the latency of GAP tasks is much higher than that of OLTP tasks, we end the execution of OLTP tasks after several rounds of GAP tasks to ensure the time of OLTP and GAP is similar in HTGAP workloads. Unless otherwise noted, we dedicate one machine for OLTP requests (OLTP server) and the other for GAP requests (GAP server). On the OLTP server, we pin 20 cores for OLTP worker threads and 1 core for the OLTP client thread. On the GAP server, we pin 12 cores for GAP worker threads, 10 cores for log replayer threads, and 1 core for the GAP client thread. The single core for clients is sufficient for request generation. We set the epoch interval to 15 milliseconds. For edge segment compaction (§5.1), we choose a strategy of compaction at edge insertion instead of periodic compaction, which can reduce repeated compaction when the OLTP throughput is high.

Benchmarks. Considering that there is no standard HTGAP benchmark, we first retrofit LDBC Social Network Benchmark (SNB) [4] and TPC-C [71] as two new HTGAP benchmarks and further select several typical graph datasets as our micro-benchmarks. The graphs are summarized in Table 1.

LDBC SNB is a GAP benchmark that contains a social network graph and different types of GAP workloads. We select to load 50% of edges and insert another 50% of edges for transactions. We set the scale factor (SF) of the dataset to 10 (about 8.4 GB) on each server.

TPC-C is a standard OLTP benchmark that contains relational datasets and five kinds of transactions. We extract two bipartite graphs (ORDER-ORDERLINE and CUSTOMER-ITEM graphs) from relational tables by mapping rows in entity tables as vertices and adding edges based on relationship tables. We deploy 20 warehouses on each server.

On the graphs of LDBC SNB and TPC-C, we choose three types of GAP workloads as prior work [29]:

- Graph analytics (GA): three representative graph algorithms from LDBC Graphalytics Benchmark [3], including PageRank (PR), Connected Components (CC), and Single Source Shortest Path (SSSP).
- Graph traversal (GT) [81]: three scan-dominated queries from LDBC SNB [4], including one interactive query (IS-3) and two business intelligence (BI) queries (BI-2 and BI-3). These queries access both the graph topology and properties.
- Graph neural network (GNN) [75, 83]: inference on three popular models, including Graph Convolution Network (GCN) [40], GraphSage (GSG) [35], and Simple Graph Convolution (SGC) [62].[1]

Comparing targets. To show the efficacy of the loosely-coupled design for HTGAP, we mainly focus on the performance of OLTP and GAP, and the freshness in GART against two different solutions (see §1). For Solution ①, we connect DrTM+H with GraphScope [29] (DH+GS), where transactions are served by DrTM+H (the same OLTP engine of GART), and transactional data are periodically loaded into GraphScope, a state-of-the-art GAP engine. For Solution ②, we evaluate Neo4j [6], a popular graph database that supports both transactions and GAP.[2]

To study the efficiency of our dynamic graph storage, we integrated LiveGraph [87], a state-of-the-art transactional graph storage, into GART (G/LG). LiveGraph uses a highly optimized adjacency list format (similar to Fig. 6(b)) to store the graph topology and a row store with fine-grained MVCC to store properties. GART's graph storage and LiveGraph provide the same interfaces. Therefore, GART outperforms G/LG solely due to three design choices of our graph storage. Note that all systems, except Neo4j, use the same OLTP and GAP engines, namely DrTM+H and GraphScope, with the same configurations (e.g., the number of worker threads) for fairness. Although we try our best to run HTGAP workloads on Neo4j and DH+GS, they still cannot support TPC-

[1]We run graph analytics and graph neural network workloads directly on the CUSTOMER-ITEM graph in the TPC-C dataset and PERSON-POST graph in the LDBC SNB dataset. Graph traversal queries from LDBC SNB are tightly coupled with its dataset. We rewrite queries on the TPC-C dataset and ensure that they have the same computation patterns as LDBC SNB.

[2]We also evaluated TigerGraph [24] as an alternative solution. However, TigerGraph's timestamp only provides second-level accuracy, which limits its ability to evaluate freshness with sub-second precision.

Table 2: *A comparison of OLTP throughput (in transactions per second), GAP latency (in milliseconds), and freshness (in milliseconds) using different workloads among* GART*, a combination of* DrTM+H *and* GraphScope *for offline data processing* (DH+GS)*,* Neo4j *(not support GNN workloads), and* GART *w/* LiveGraph (G/LG)*. Note that ↓ (resp. ↑) indicates low (resp. high) is better.*

Workloads		LDBC SNB				TPC-C	
		GART	DH+GS	Neo4j	G/LG	GART	G/LG
OLTP ↑		1837 K	1929 K	3.5 K	1836 K	245 K	212 K
GA ↓	PR	377	309	5323	1276	204	329
	CC	362	312	4726	1137	210	300
	SSSP	513	433	4668	1381	315	410
GT ↓	IS-3	17.9	16.9	2.0	18.0	14.2	14.6
	BI-2	235	201	568	828	1884	2806
	BI-3	292	266	573	1278	266	586
GNN ↓	GCN	1097	940	×	1834	623	636
	GSG	1774	1443	×	2502	386	418
	SGC	779	717	×	1237	184	257
Freshness ↓		18	15683	5	25	18	25

C due to costly code transcription and graph extraction from complex logs, respectively.

Coding effort. To extract graph data from relational data, we only write about 10 LoCs for each benchmark (LDBC SNB and TPC-C), thanks to graph extraction interfaces in GART (§4.1). This is far less code than OLTP and GAP programs, which can be inherited directly from existing specific systems. In contrast, we write 584 LoCs in DH+GS for loading graph data and 70 LoCs in Neo4j for rewriting transactions.

6.2 Overall Performance

We first show the overall performance of all baselines for HTGAP workloads in Table 2. We use transaction throughput as the evaluation metric for OLTP workloads, and computation latency (execution time) for GAP workloads. We also evaluate the freshness of each system, namely, the maximum time delay between an update was committed in OLTP and this update is visible in GAP workloads [65]. In a nutshell, among all baselines, only GART can simultaneously satisfy the requirements of *performance* and *freshness*.

OLTP performance. For systems based on the loosely-coupled design (GART, G/LG) and DH+GS, OLTP throughput is not impacted by GAP workloads. The peak throughput of GART can reach over 1,837,000 and 245,000 transactions per second on datasets LDBC SNB and TPC-C, respectively. GART performs 2-3 orders of magnitude better than Neo4j (Solution ②). Neo4j has the lowest OLTP performance as the graph data model is less efficient than the relation model for OLTP workloads. GART and G/LG show an OLTP throughput reduction of only 5% compared to DrTM+H with offline data processing (DH+GS). This result demonstrates that the loosely-coupled design can support HTGAP without necessarily sacrificing OLTP performance.

GAP performance. Among all baselines, DH+GS achieves the best GAP performance, as its graph data is stored as a *static* graph, which uses compact graph representation without concurrency control (e.g., MVCC). However, its freshness is extremely high. Neo4j performs much worse than GART and G/LG due to its adjacency-list-based storage [68, 87]. GART greatly outperforms G/LG except for the IS-3 query, thanks to our dynamic graph storage which takes the characteristic of HTGAP workloads into consideration (breakdown details in §6.3). The IS-3 query only involves a very limited graph data size, thus the overall execution time is dominated by cross-language invocation overheads. The backend and frontend engines of GraphScope are developed with Rust and C++, respectively.

Freshness. The freshness of GART is about 18 ms, which is much lower than most GAP latencies and is independent of datasets. It is three orders of magnitude better than DH+GS (Solution ①). Similar to VEGITO [65], the freshness of GART is only determined by the epoch interval (15 ms). DH+GS has the worst freshness (more than 15 seconds), which is unaffected by the ETL frequency and depends on the graph data size. This is because the immutable graph storage requires the entire graph to be reloaded whenever changes are made. The freshness of G/LG is 25 ms due to the lower write performance of the graph storage. The freshness of Neo4j is only about 5 ms, as data is committed in place.

6.3 Breakdown Analysis on GAP Performance

From Table 2, we observe that GART achieves a large performance improvement over other systems on graph analytics and traversal workloads and a relatively small improvement on GNN workloads. To gain a deeper understanding of the dynamic graph storage in GART, we perform a detailed comparison with G/LG.

We split a single execution of a GAP query into three parts: (P1) accessing the graph topology; (P2) getting properties from visited vertices or edges; and (P3) computation over the graph topology and properties obtained from (P1) and (P2) parts. Meanwhile, the performance gain of GART mainly comes from three aspects: (A1) the efficient and mutable CSR with good locality of edge scan; (A2) the coarse-grained MVCC that alleviates the costly versioning; and (A3) the flexible property storage that adapts to access patterns. Table 3 shows the results of a breakdown analysis of three GAP workloads for the LDBC SNB dataset. Since G/LG and GART have the same backend GAP engine, their performance in part (P3) is nearly identical. Meanwhile, the design differences between (A1) and (A2) affect the performance of the (P1) part, while the performance of the (P2) part is influenced by (A3).

Graph analytics. Table 2 shows the execution time of dif-

Table 3: *Breakdown analysis of G/LG and GART over three GAP workloads (in milliseconds) using the LDBC dataset.*

	Storage	Topo (S1)	Prop (S2)	Comp (S3)	Total
PR	GART	107 (28%)	163 (44%)	107 (28%)	377
	G/LG	658 (52%)	486 (38%)	132 (10%)	1276
BI-3	GART	10 (3%)	178 (61%)	104 (36%)	292
	G/LG	40 (3%)	1115 (87%)	123 (10%)	1278
SGC	GART	36 (5%)	477 (61%)	266 (34%)	779
	G/LG	236 (19%)	732 (59%)	269 (22%)	1237

ferent algorithms and excludes the time of storing outputs. The latency of PageRank, CC, and SSSP on G/LG is 2.5×, 2.3×, and 2.0× higher than GART, respectively. The performance gain of GART on graph analytics workloads is mainly from the better performance of accessing the dynamic graph topology due to (A1) and (A2), and the higher efficiency of obtaining required properties thanks to (A3). For example, as shown in Table 3, the end-to-end execution time of PageRank is dominated by the graph traversal (edge scan) operations (P1) and getting required properties (P2). GART's graph storage outperforms G/LG by 6.2× in (P1), with 82% of the improvement attributed to (A1). GART also outperforms G/LG by 3.0× in (P2) due to (A3). Note that PageRank uses only one property, so GART does not group it with others.

Graph traversal. For graph traversal workloads, GART's latency is 3.5× and 4.4× lower than G/LG's for two BI queries (BI-2 and BI-3) on the LDBC SNB dataset, while GART performs almost as well as its competitor on the interactive query (IS-3). The reason is that compared with BI queries, the interactive query only involves a very limited graph data size (vertices and edges as well as their properties), and our optimized storage design cannot contribute much in such a circumstance. Instead, the computations of two BI queries rely on scanning one or several properties of a large number of vertices or edges, and (A1) to (A3) can benefit a lot. Observe that given a BI query (BI-3), compared with G/LG, GART performs 4.0× and 6.3× faster on accessing the graph topology (P1) and obtaining required properties (P2) parts, respectively (see Table 3).

Graph neural networks. On three GNN workloads, GART outperforms G/LG by 1.3× on average. GNN workloads need several/all properties of vertices/edges as raw features to conduct GNN computation, and (A3) allows users to store and get required properties in a more efficient way. As we can see from Table 3, on SGC, GART is 1.5× faster than G/LG in obtaining the required properties in (P2). Meanwhile, (A1) and (A2) make GART 6.6× faster than G/LG in (P1). The performance improvements over the TPC-C dataset are relatively small, because the CUSTOMER-ITEM graph in dataset TPC-C is very dense, and the end-to-end execution time of GNN workloads is dominated by the complex computations over properties of vertices or edges (P3).

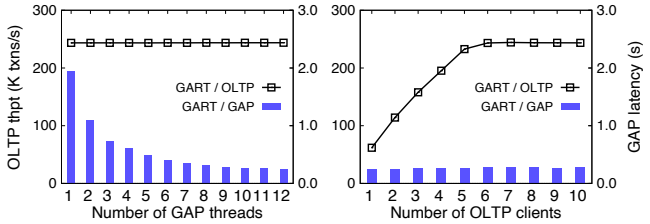

Fig. 9. *Performance isolation on GART with the increase of (a) GAP workloads and (b) OLTP clients, respectively.*

6.4 Performance Isolation

To demonstrate the performance isolation in the HTGAP workloads, we evaluate the OLTP and GAP performance with the increase of GAP and OLTP clients. We use the TPC-C benchmark as the OLTP workloads and execute PageRank on the CUSTOMER-ITEM graph derived from the TPC-C schema. We use the number of OLTP clients and the number of worker threads for a single GAP request to control the workloads.

In general, GART provides strong performance isolation between OLTP and GAP workloads. Fig. 9(a) shows the performance of OLTP and GAP workloads when we gradually increase the number of GAP worker threads. The OLTP performance degradation is trivial (1%), even if the GAP workloads are saturated. This is due to the physical isolation in GART, as GAP workloads do not interfere with transactions. On the other hand, when we increase the number of OLTP clients, as shown in Fig. 9(b), the performance degradation of GAP workloads is about 12%. This is because the number of edge versions also increases, causing additional overhead to check versions. At 5 clients, the OLTP server's maximum capacity is reached due to a fixed number of cores being allocated for its use.

6.5 Graph Topology Storage

Edge scan is a common and costly operation in GAP workloads, such as PageRank and LPA. To study how the performance of edge scan is affected by different graph topology storages, we compare the following typical graph storages.

- **CSR** is a compact structure without the support of updates, which is widely used by the *static* graph topology storages.
- **LiveGraph** [87] is a state-of-the-art dynamic graph storage that uses adjacency lists and fine-grained MVCC.
- **SegCSR** is a CSR-like topology storage proposed by GART, which uses segment-based design and coarse-grained MVCC. Note that each edge segment has a 1 MB initial size and manages 4,096 vertices.
- **SegCSR/TS** is similar to SegCSR, except that it uses fine-grained MVCC as LiveGraph.

To simulate diverse workloads, we load graphs in two patterns: (1) *bulk load*, i.e., edges are sorted by their source vertices and loaded sequentially, and (2) *random insertion*, i.e., edges are loaded randomly to simulate the behavior in trans-

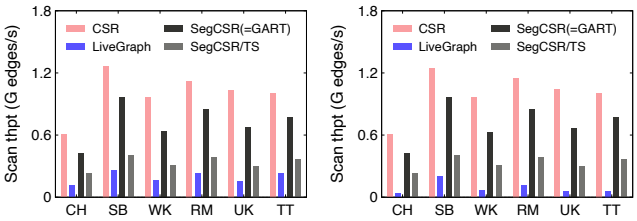

Fig. 10. *Comparison of single-version edge scan perf. for different topology storages using (a) bulk load and (b) random insertion.*

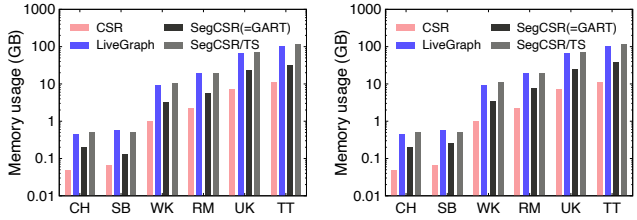

Fig. 11. *Comparison of memory usage (in log scale) for different topology storages using (a) bulk load and (b) random insertion.*

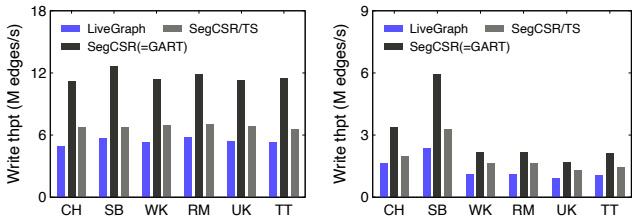

Fig. 12. *Comparison of write throughput for different topology storages using (a) bulk load and (b) random insertion.*

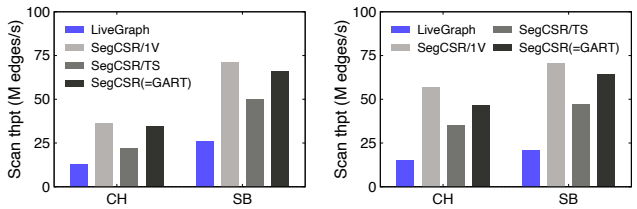

Fig. 13. *Comparison of multi-version edge scan perf. for different topology storages using (a) bulk load and (b) random insertion.*

actions. We scan edges of each vertex and evaluate edges read per second as scan throughput.

Read-only workloads. We first evaluate the performance of the single-version edge scan without version checking. As shown in Fig. 10(a), with the bulk load, SegCSR exhibits consistent performance behavior across various datasets. For example, SegCSR only incurs a 35% slowdown compared to CSR for WK, while LiveGraph incurs more than an 80% slowdown. Compared with adjacency lists for each vertex in LiveGraph, SegCSR has a better locality and fully exploits CPU prefetching as it associates the edges of many vertices in a segment. Moreover, the edge scan throughput of SegCSR is more than 2× that of SegCSR/TS since SegCSR adopts a simpler data structure for edges. With random insertion, the locality of LiveGraph suffers from memory allocation. As shown in Fig. 10(b), SegCSR outperforms LiveGraph by up to 12.5× (from 4.7×) and only incurs about a 30% slowdown compared to CSR.

The memory usage is shown in Fig. 11. Compared with CSR, SegCSR requires about 3× memory of CSR for updates, which is significantly less than that of LiveGraph (8.8× of CSR). The coarse-grained MVCC helps SegCSR reduce memory largely by using the epoch table for each vertex instead of timestamps for each edge. Compared to SegCSR/TS, SegCSR reduces memory usage by up to 3.8×. The memory usage of SegCSR/TS is higher than that of LiveGraph (typically less than 18%) due to the free slots in the edge segments.

Read-write workloads. We further evaluate the performance of the write and the multi-version edge scan. We scan the edges in latest stable version via latest_repoch (§2) when a dedicated number of edges have been inserted. For the bulk load setting, we use the ORDER-ORDERLINE graph in TPC-C and PERSON-POST graph in LDBC SNB. With the random insertion setting, we utilize a shuffled CUSTOMER-ORDER graph in TPC-C and PERSON-POST graph in LDBC SNB.

As shown in Fig. 12, the write throughput of SegCSR is about 2.2× and 2× of LiveGraph with the bulk load and random insertion, respectively. According to the performance of SegCSR/TS, about 70% of the performance improvement is due to the fact that the coarse-grained MVCC of GART writes the newly generated version (epoch) number to the epoch table only once, instead of writing the version number on every update. Moreover, writes of SegCSR do not perform costly GC-related operations, unlike with LiveGraph. SegCSR will copy edges from an old segment with insufficient space to a new segment with a larger space. It introduces high tail latency (more than 7,000× of ordinary edge insertion latency on average) in edge insertion, but the frequency of it being triggered is less than 0.01%.

As shown in Fig. 13(a), with the bulk load setting, read performance of SegCSR outperforms LiveGraph by 2.5× due to better locality and coarse-grained MVCC. It is very close to the upper bound (single-version read performance, SegCSR/1V), while LiveGraph is about 37% of SegCSR/1V. To show the efficiency of coarse-grained MVCC, SegCSR/TS outperforms LiveGraph only by 1.7×. With the random insertion setting shown in Fig. 13(b), SegCSR and SegCSR/TS outperform LiveGraph by 3.1× and 2.3×, respectively. It indicates that the coarse-grained MVCC in GART can largely increase read performance for dynamic workloads.

6.6 Flexible Property Storage

To study the performance of the flexible property storage, we compared it with two typical property storages: row store (row) and column store (col). For the read (resp. write) performance, we scan (resp.update) the properties of each vertex using CUSTOMER vertices derived from TPC-C. We control the number of columns scanned and updated, and evalu-

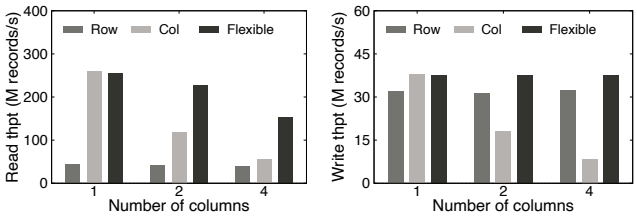

Fig. 14. *Comparison of (a) read throughput and (b) write throughput for different property storages with different numbers of columns read and written, respectively.*

ate the number of processed vertices per second as read and write throughput, respectively.

Fig. 14(a) reports the read throughput. The performance of the row-based property storage is fixed as the number of scanned columns increases since it needs to fetch at least one cache line. However, the performance of column-based property storage significantly drops due to cross-column access. Compared with the row-based storage, the flexible property storage achieves better performance, especially when scanning a few columns. For example, the flexible property storage achieves 5.9× read throughput when scanning only one column. The write operations of the flexible property storage also outperform existing storage models, as shown in Fig. 14(b). It achieves a speedup of 1.2× and 4.4× compared to row-based and column-based storages, respectively, when writing four columns of properties.

Row-based storages and column-based storages have different performance behaviors for reads and writes. We find that the read operation is light and dominated by the memory footprint, while the write operation is dominated by the number of writes due to the overhead of memory copy and atomic operations. The flexible property storage allows users to combine attributes into a column family *on-the-fly* with some overhead. In our experiments, it takes about 1.1 seconds to create a property snapshot with 4 columns as a column family for 229 MB properties.

7 Related Work

HTAP systems. HTAP systems have three main typical design choices. Dual systems [51, 57, 58, 82] combine two specialized systems for OLTP and OLAP scenarios, while single-layout systems [39, 60, 64] support HTAP workloads from either an OLTP or an OLAP system. Dual-layout systems [9, 14, 15, 19, 44, 50] aim to build a single system with different data layouts for the two scenarios. Vegito [65] is proposed to retrofit fault-tolerant backups to support hybrid workloads, which arrives at a sweet spot for the performance-freshness tradeoff. These works are developed for relational data, while GART extends Vegito to constantly maintain a graph layout for transactional data from an OLTP system to support dynamic graph analytical processing.

Graph databases. Graph databases [2, 6, 8, 27] support both OLTP and GAP in a single system. In order to conduct efficient graph updates, they typically adopt linked lists to store adjacency lists, which downgrades the performance of edge scan and the whole GAP workloads. LiveGraph [87] devises the Transactional Edge Logs (TELs) based on adjacency lists to support both efficient sequential scan and edge insertion. Adding a CSR-based in-memory property graph representation in a relational database has been investigated by Oracle, but without support for updates [13]. GART decouples OLTP and GAP execution to make both workloads more efficient.

Dynamic graph systems. Prior work presents many general-purpose dynamic graph systems [24, 30, 41]. Terrace [54] uses PMA [25, 80], a dynamic memory array based on tree-based index structures, to store edges of streaming graphs. CSR++ [32] combines segmented vertex arrays and vector-based adjacency lists for each vertex, which does not guarantee the locality of edge scan from adjacent vertices (see Fig. 6(b)). Moreover, CSR++ does not support multi-versioning. Sortledton [33] provides a general-purpose and transactional graph data structure based on adjacency lists. Teseo [26] and LLAMA [49] are also CSR-like and guarantee the locality of edge scan. While general-purpose dynamic graph storages can replace GART's storage in functions, they may face performance issues. For example, GART could use LLAMA [49] as the graph storage, but it would have to copy data pages for each snapshot, which would be inefficient for scenarios with high data generation rates or extreme freshness. Based on insights from HTGAP, the graph storage of GART does not need to support full transactional semantics, allowing for new designs and optimizations in the topology storage. In addition, GART also provides the flexible property storage for diverse GAP workloads.

8 Conclusion

This paper presents GART, the first hybrid transactional and graph analytical processing (HTGAP) system based on a loosely-coupled design. It proposes expressive interfaces for transparent data conversion and an efficient dynamic graph storage with good locality. Evaluations confirm its efficacy and efficiency. The source code of GART, including all benchmarks, is available at https://github.com/SJTU-IPADS/vegito/tree/gart.

Acknowledgment

We sincerely thank our shepherd Jean-Pierre Lozi and the anonymous reviewers for their insightful comments and feedback. This work was supported in part by the National Natural Science Foundation of China (No. 62272291, 61925206), the Fundamental Research Funds for the Central Universities, the HighTech Support Program from Shanghai Committee of Science and Technology (No. 22511106200), and a research grant from Alibaba Group through the Alibaba Innovative Research Program. Corresponding author: Rong Chen (rongchen@sjtu.edu.cn).

REFERENCES

[1] erwin Data Modeler. https://www.erwin.com/.

[2] JanusGraph. https://janusgraph.org/.

[3] LDBC Graphalytics. https://ldbcouncil.org/benchmarks/graphalytics/.

[4] LDBC Social Network Benchmark (LDBC-SNB). https://ldbcouncil.org/benchmarks/snb/.

[5] Navicat. https://navicat.com/.

[6] Neo4j. https://neo4j.com/.

[7] Neptune. https://aws.amazon.com/neptune/.

[8] OrientDB. http://orientdb.com/.

[9] Ioannis Alagiannis, Stratos Idreos, and Anastasia Ailamaki. H2O: A hands-free adaptive store. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, page 1103–1114, 2014.

[10] Alibaba Cloud. Double 11 real-time monitoring system with time series database. https://www.alibabacloud.com/blog/594855, 2019.

[11] Renzo Angles. The property graph database model. In *AMW*, 2018.

[12] Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Computing Surveys (CSUR)*, 40(1):1–39, 2008.

[13] Marco Arnaboldi, Jean-Pierre Lozi, Laurent Phillipe Daynes, Vlad Ioan Haprian, Shasank Kisan Chavan, Kapp Hugo, and Sungpack Hong. Parallel and efficient technique for building and maintaining a main memory, CSR-based graph index in an RDBMS, August 17 2021. US Patent 11,093,459.

[14] Joy Arulraj, Andrew Pavlo, and Prashanth Menon. Bridging the archipelago between row-stores and column-stores for hybrid workloads. In *Proceedings of the 2016 International Conference on Management of Data*, pages 583–598, 2016.

[15] Martin Boissier. Reducing the footprint of main memory HTAP systems: Removing, compressing, tiering, and ignoring data. In *Proceedings of the VLDB 2018 PhD Workshop*, 2018.

[16] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. UbiCrawler: A scalable fully distributed web crawler. *Software: Practice and Experience*, 34(8):711–726, 2004.

[17] Shaosheng Cao, XinXing Yang, Cen Chen, Jun Zhou, Xiaolong Li, and Yuan Qi. TitAnt: Online real-time transaction fraud detection in Ant Financial. *Proceedings of the VLDB Endowment*, 12(12):2082—-2093, August 2019.

[18] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-MAT: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*, pages 442–446, 2004.

[19] Jianjun Chen, Yonghua Ding, Ye Liu, Fangshi Li, Li Zhang, Mingyi Zhang, Kui Wei, Lixun Cao, Dan Zou, Yang Liu, Lei Zhang, Rui Shi, Wei Ding, Kai Wu, Shangyu Luo, Jason Sun, and Yuming Liang. ByteHTAP: ByteDance's HTAP system with high data freshness and strong data consistency. *Proc. VLDB Endow.*, 15(12):3411–3424, 2022.

[20] Peter P. Chen. The entity-relationship model—toward a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36, 1976.

[21] Rong Chen and Haibo Chen. Wukong: A distributed framework for fast and concurrent graph querying. *ACM SIGOPS Operating Systems Review*, 55(1):77–83, 2021.

[22] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. PowerLyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems*, Eurosys '15, pages 1–15, 2015.

[23] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. Fast and general distributed transactions using RDMA and HTM. In *Proceedings of the Eleventh European Conference on Computer Systems*, Eurosys '16, page 26, 2016.

[24] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. Kineograph: Taking the pulse of a fast-changing and connected world. In *Proceedings of the 7th ACM European Conference on Computer Systems*, Eurosys '12, page 85–98, 2012.

[25] Dean De Leo and Peter Boncz. Packed memory arrays - rewired. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 830–841, 2019.

[26] Dean De Leo and Peter Boncz. Teseo and the analysis of structural dynamic graphs. *Proceedings of the VLDB Endowment*, 14(6):1053–1066, 2021.

[27] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor Lee. TigerGraph: A native MPP graph database. *arXiv preprint arXiv:1901.08248*, 2019.

[28] Christian Fahrner and Gottfried Vossen. A survey of database design transformations based on the entity-relationship model. *Data Knowl. Eng.*, 15(3):213–250, 1995.

[29] Wenfei Fan, Tao He, Longbin Lai, Xue Li, Yong Li, Zhao Li, Zhengping Qian, Chao Tian, Lei Wang, Jingbo Xu, et al. GraphScope: A unified engine for big graph processing. *Proceedings of the VLDB Endowment*, 14(12):2879–2892, 2021.

[30] Guanyu Feng, Zixuan Ma, Daixuan Li, Shengqi Chen, Xiaowei Zhu, Wentao Han, and Wenguang Chen. RisGraph: A real-time streaming system for evolving graphs to support sub-millisecond per-update analysis at millions ops/s. In *Proceedings of the 2021 International Conference on Management of Data*, pages 513–527, 2021.

[31] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *Proceedings of the 7th International Conference on Learning Representations*, ICLR '19, 2019.

[32] Soukaina Firmli, Vasileios Trigonakis, Jean-Pierre Lozi, Iraklis Psaroudakis, Alexander Weld, Dalila Chiadmi, Sungpack Hong, and Hassan Chafi. CSR++: A fast, scalable, update-friendly graph data structure. In *24th International Conference on Principles of Distributed Systems*, OPODIS '20, 2020.

[33] Per Fuchs, Domagoj Margan, and Jana Giceva. Sortledton: A universal, transactional graph data structure. *Proceedings of the VLDB Endowment*, 15(6):1173–1186, 2022.

[34] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. GraphX: Graph processing in a distributed dataflow framework. In *11th USENIX symposium on operating systems design and implementation*, OSDI '14, pages 599–613, 2014.

[35] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. *Advances in neural information processing systems*, 30, 2017.

[36] Henry Haselgrove. Wikipedia page-to-page link database. http://haselgrove.id.au/wikipedia.htm, 2010.

[37] Mohamed S Hassan, Tatiana Kuznetsova, Hyun Chai Jeong, Walid G Aref, and Mohammad Sadoghi. Extending in-memory relational database engines with native graph support. In *Proceedings of the 21st International Conference on Extending Database Technology*, EDBT '18, pages 25–36, 2018.

[38] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. TiDB: A Raft-based HTAP database. *Proc. VLDB Endow.*, 13(12):3072–3084, August 2020.

[39] Alfons Kemper, Thomas Neumann, Florian Funke, Viktor Leis, and Henrik Mühe. HyPer: Adapting columnar main-memory data management for transactional and query processing. *IEEE Data Eng. Bull.*, 35(1):46–51, 2012.

[40] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*, ICLR '17, 2017.

[41] Pradeep Kumar and H Howie Huang. GraphOne: A data store for real-time analytics on evolving graphs. *ACM Transactions on Storage (TOS)*, 15(4):1–40, 2020.

[42] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a social network or a news media? In *Proceedings of the 19th international conference on World wide web*, pages 591–600, 2010.

[43] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. GraphChi: Large-Scale graph computation on just a PC. In *10th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '12, pages 31–46, 2012.

[44] Per-Ake Larson, Adrian Birka, Eric N. Hanson, Weiyun Huang, Michal Nowakiewicz, and Vassilis Papadimos. Real-Time analytical processing with SQL Server. *Proc. VLDB Endow.*, 8(12):1740–1751, August 2015.

[45] Juchang Lee, SeungHyun Moon, Kyu Hwan Kim, Deok Hoe Kim, Sang Kyun Cha, and Wook-Shin Han. Parallel replication across formats in SAP HANA for scaling out mixed OLTP/OLAP workloads. *Proc. VLDB Endow.*, 10(12):1598–1609, August 2017.

[46] Chaojie Li, Wensen Jiang, Yin Yang, Shirui Pan, Gang Huang, and Lijie Guo. Predicting best-selling new products in a major promotion campaign through graph convolutional networks. *IEEE Transactions on Neural Networks and Learning Systems*, 2022.

[47] Feifei Li. Cloud-native database systems at Alibaba: Opportunities and challenges. *Proceedings of the VLDB Endowment*, 12(12):2263–2272, 2019.

[48] Hongbin Ma, Bin Shao, Yanghua Xiao, Liang Jeff Chen, and Haixun Wang. G-SQL: Fast query processing via graph exploration. *Proceedings of the VLDB Endowment*, 9(12):900–911, 2016.

[49] Peter Macko, Virendra J Marathe, Daniel W Margo, and Margo I Seltzer. LLAMA: Efficient graph analytics using large multiversioned arrays. In *2015 IEEE 31st International Conference on Data Engineering*, pages 363–374, 2015.

[50] Darko Makreshanski, Jana Giceva, Claude Barthels, and Gustavo Alonso. BatchDB: Efficient isolated execution of hybrid OLTP+OLAP workloads for interactive applications. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 37–50, 2017.

[51] Daniel Martin, Oliver Koeth, Johannes Kern, and Iliyana Ivanova. Near real-time analytics with IBM DB2 analytics accelerator. In *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT '13, pages 579–588, 2013.

[52] MySQL. MySQL internals manual: Chapter 20 the binary log. https://dev.mysql.com/doc/internals/en/binary-log.html.

[53] Sen Pan, Menghan Xu, Pei Yang, Lipeng Zhu, Aihua Zhou, and Jing Jiang. Research on application scenarios of HTAP in distribution network. In *2021 IEEE Sustainable Power and Energy Conference (iSPEC)*, pages 3916–3920, 2021.

[54] Prashant Pandey, Brian Wheatman, Helen Xu, and Aydin Buluc. Terrace: A hierarchical graph container for skewed dynamic graphs. In *Proceedings of the 2021 International Conference on Management of Data*, pages 1372–1385, 2021.

[55] Marcus Paradies, Cornelia Kinder, Jan Bross, Thomas Fischer, Romans Kasperovics, and Hinnerk Gildhoff. GraphScript: Implementing complex graph algorithms in SAP HANA. In *Proceedings of The 16th International Symposium on Database Programming Languages*, pages 1–4, 2017.

[56] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. Real-Time constrained cycle detection in large dynamic graphs. *Proc. VLDB Endow.*, 11(12):1876–1888, 2018.

[57] Jags Ramnarayan, Barzan Mozafari, Sumedh Wale, Sudhir Menon, Neeraj Kumar, Hemant Bhanawat, Soubhik Chakraborty, Yogesh Mahajan, Rishitesh Mishra, and Kishor Bachhav. SnappyData: A hybrid transactional analytical store built on Spark. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 2153–2156, 2016.

[58] Aunn Raza, Periklis Chrysogelos, Angelos Christos Anadiotis, and Anastasia Ailamaki. Adaptive HTAP through elastic resource scheduling. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 2043–2054, 2020.

[59] Noa Roy-Hubara, Lior Rokach, Bracha Shapira, and Peretz Shoval. Modeling graph database schema. *IT Prof.*, 19(6):34–43, 2017.

[60] Mohammad Sadoghi, Souvik Bhattacherjee, Bishwaranjan Bhattacharjee, and Mustafa Canim. L-Store: A real-time OLTP and OLAP system. In *Proceedings of the 21th International Conference on Extending Database Technology*, EBDT '18, 2018.

[61] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. The ubiquity of large graphs and surprising challenges of graph processing. *Proc. VLDB Endow.*, 11(4):420–431, dec 2017.

[62] Michael Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. Modeling relational data with graph convolutional networks. In *European semantic web conference*, pages 593–607, 2018.

[63] Alex Shamis, Matthew Renzelmann, Stanko Novakovic, Georgios Chatzopoulos, Aleksandar Dragojević, Dushyanth Narayanan, and Miguel Castro. Fast general distributed transactions with Opacity. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 433–448, 2019.

[64] Ankur Sharma, Felix Martin Schuhknecht, and Jens Dittrich. Accelerating analytical processing in MVCC using fine-granular high-frequency virtual snapshotting. In *Proceedings of the 2018 International Conference on Management of Data*, pages 245–258, 2018.

[65] Sijie Shen, Rong Chen, Haibo Chen, and Binyu Zang. Retrofitting high availability mechanism to tame hybrid transaction/analytical processing. In *15th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '21, pages 219–238, 2021.

[66] Sijie Shen, Xingda Wei, Rong Chen, Haibo Chen, and Binyu Zang. DrTM+B: Replication-driven live reconfiguration for fast and general distributed transaction processing. *IEEE Transactions on Parallel and Distributed Systems*, 33(10):2628–2643, 2022.

[67] Jiaxin Shi, Youyang Yao, Rong Chen, Haibo Chen, and Feifei Li. Fast and concurrent RDF queries with RDMA-based distributed graph exploration. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI '16, pages 317–332, 2016.

[68] John Stegeman. Native vs. non-native graph database. https://neo4j.com/blog/native-vs-non-native-graph-technology/.

[69] Wen Sun, Achille Fokoue, Kavitha Srinivas, Anastasios Kementsietsidis, Gang Hu, and Guo Tong Xie. SQLGraph: An efficient relational-based property graph store. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, 2015.

[70] Qiaoyu Tan, Ninghao Liu, Xing Zhao, Hongxia Yang, Jingren Zhou, and Xia Hu. Learning to hash with graph neural networks for recommender systems. In *Proceedings of The Web Conference 2020*, pages 1988–1998, 2020.

[71] The Transaction Processing Council. TPC-C benchmark v5.11. http://www.tpc.org/tpcc/.

[72] Yuanyuan Tian, En Liang Xu, Wei Zhao, Mir Hamid Pirahesh, Sui Jun Tong, Wen Sun, Thomas Kolanko, Md Shahidul Haque Apu, and Huijuan Peng. IBM DB2 Graph: Supporting synergistic and retrofittable graph queries inside IBM DB2. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 345–359, 2020.

[73] Vasileios Trigonakis, Jean-Pierre Lozi, Tomáš Faltín, Nicholas P. Roth, Iraklis Psaroudakis, Arnaud Delamare, Vlad Haprian, Calin Iorgulescu, Petr Koupy, Jinsoo Lee, Sungpack Hong, and Hassan Chafi. aDFS: An almost depth-first-search distributed graph-querying system. In *2021 USENIX Annual Technical Conference*, USENIX ATC '21, pages 209–224, 2021.

[74] Jizhe Wang, Pipei Huang, Huan Zhao, Zhibo Zhang, Binqiang Zhao, and Dik Lun Lee. Billion-scale commodity embedding for e-commerce recommendation in Alibaba. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 839–848, 2018.

[75] Lei Wang, Qiang Yin, Chao Tian, Jianbang Yang, Rong Chen, Wenyuan Yu, Zihang Yao, and Jingren Zhou. FlexGraph: a flexible and efficient distributed framework for GNN training. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 67–82, 2021.

[76] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. Deep Graph Library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315*, 2019.

[77] Rui Wang, Yongkun Li, Hong Xie, Yinlong Xu, and John CS Lui. GraphWalker: An I/O-efficient and resource-friendly graph analytic system for fast and scalable random walks. In *2020 USENIX Annual Technical Conference*, USENIX ATC '20, pages 559–571, 2020.

[78] Xingda Wei, Rong Chen, Haibo Chen, Zhaoguo Wang, Zhenhan Gong, and Binyu Zang. Unifying timestamp with transaction ordering for MVCC with decentralized scalar timestamp. In *Proceedings of 18th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '21, 2021.

[79] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. Deconstructing RDMA-enabled distributed transactions: Hybrid is better! In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '18, pages 233–251, 2018.

[80] Brian Wheatman and Randal Burns. Streaming sparse graphs using efficient dynamic sets. In *2021 IEEE International Conference on Big Data (Big Data)*, pages 284–294, 2021.

[81] Xiating Xie, Xingda Wei, Rong Chen, and Haibo Chen. Pragh: Locality-preserving graph traversal with split live migration. In *Proceedings of the 2019 USENIX Annual Technical Conference*, USENIX ATC '19, pages 723–738, 2019.

[82] Jiacheng Yang, Ian Rae, Jun Xu, Jeff Shute, Zhan Yuan, Kelvin Lau, Qiang Zeng, Xi Zhao, Jun Ma, Ziyang Chen, Yuan Gao, Qilin Dong, Junxiong Zhou, Jeremy Wood, Goetz Graefe, Jeff Naughton, and John Cieslewicz. F1 Lightning: HTAP as a service. *Proc. VLDB Endow.*, 13(12):3313–3325, August 2020.

[83] Jianbang Yang, Dahai Tang, Xiaoniu Song, Lei Wang, Qiang Yin, Rong Chen, Wenyuan Yu, and Jingren Zhou. GNNLab: A factored system for sample-based GNN training over GPUs. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, pages 417–434, 2022.

[84] Kangfei Zhao and Jeffrey Xu Yu. All-in-One: Graph processing in RDBMSs revisited. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 1165–1180, 2017.

[85] Ai-Hua Zhou, Li-Peng Zhu, Meng-Han Xu, Sen Pan, Jun-Feng Qiao, Hong-Bin Qiu, and Song Deng. Research on mixed transaction analytical data management oriented to data middle platform. In *2021 IEEE International Conference on Progress in Informatics and Computing (PIC)*, pages 308–312, 2021.

[86] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A computation-centric distributed graph processing system. In *12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '16, pages 301–316, 2016.

[87] Xiaowei Zhu, Guanyu Feng, Marco Serafini, Xiaosong Ma, Jiping Yu, Lei Xie, Ashraf Aboulnaga, and Wenguang Chen. LiveGraph: A transactional graph storage system with purely sequential adjacency list scans. *Proceedings of the VLDB Endowment*, 13(7):1020–1034, 2020.