G-Tran: A High Performance Distributed Graph Database with a Decentralized Architecture

Hongzhi Chen, Changji Li, Chenguang Zheng, Chenghuan Huang, Juncheng Fang, James Cheng, Jian Zhang
{hzchen, cjli, cgzheng, chhuang, jcfang6, jcheng, jzhang}@cse.cuhk.edu.hk
Department of Computer Science and Engineering, The Chinese University of Hong Kong

ABSTRACT

Graph transaction processing poses unique challenges such as random data access due to the irregularity of graph structures, low throughput and high abort rate due to the relatively large read/write sets in graph transactions. To address these challenges, we present G-Tran, a remote direct memory access (RDMA)-enabled distributed in-memory graph database with serializable and snapshot isolation support. First, we propose a graph-native data store to achieve good data locality and fast data access for transactional updates and queries. Second, G-Tran adopts a fully decentralized architecture that leverages RDMA to process distributed transactions with the massively parallel processing (MPP) model, which can achieve high performance by utilizing all computing resources. In addition, we propose a new multi-version optimistic concurrency control (MV-OCC) protocol with two optimizations to address the issue of large read/write sets in graph transactions. Extensive experiments show that G-Tran achieves competitive performance compared with other popular graph databases on benchmark workloads.

PVLDB Reference Format:
Hongzhi Chen, Changji Li, Chenguang Zheng, Chenghuan Huang, Juncheng Fang, James Cheng, Jian Zhang. G-Tran: A High Performance Distributed Graph Database with a Decentralized Architecture. PVLDB, 15(11): 2545 - 2558, 2022.
doi:10.14778/3551793.3551813

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/yaobaiwei/GTran.

1 INTRODUCTION

Graph data are abundant today in both industrial applications and academic research. In order to support efficient graph data storage and management, graph databases have become an essential infrastructure. However, most existing graph databases [1, 2, 4, 8, 9, 13, 26, 31, 35, 43] have shortcomings in their designs or functionalities that lead to performance bottlenecks in the processing of graph queries and transactions. For example, JanusGraph [2] uses BigTable [16] model, OrientDB [9] and ArangoDB [4] use Multi-Model store, and A1 [15] uses key-value store, to represent graphs respectively. That will lead to significant read/write amplifications during the graph queries and updates. Neo4J [8] and TigerGraph [26] call themselves *native graph stores*, as to distinguish from the non-native stores (e.g., relational, columnar and key-value stores). We cannot find the details of TigerGraph's native graph store design as it is not open-source. Neo4j represents the edges and properties of every vertex using double linked lists, which is efficient for graph updates such as edge addition/deletion, but it also incurs overheads for random access when a graph query needs to visit the edges and properties of a vertex.

In addition to handling graph-specific read/write, transaction isolation is another important functionality in DBMSs, which controls how transaction integrity is visible to concurrent users in order to maintain the correctness of transaction processing. Transaction isolation (ideally strict serializability) is challenging when dealing with large-scale graph data. Read-only graph transactions tend to have large *read sets* since a large number of vertices and edges can be easily involved after just two to three hops of traversal starting from a vertex, as most real-world graphs exhibit a power-law degree distribution. Similarly, read-write transactions may also have large *write sets* (see Figure 10). These large read/write sets lead to high contention in concurrent transaction processing. As graph transactions have relatively long processing time, consequently the contention becomes more serious, which in turn leads to high abort rate and low throughput.

These unique challenges in graph transaction processing motivate us to design a new distributed graph database system, called *G-Tran*, for high-performance transaction processing on property graphs [10]. To the best of our knowledge, G-Tran is the first RDMA-enabled graph database that provides strong consistency, i.e., *serializability* (*SR*) and *snapshot isolation* (*SI*), low latency and high throughput for graph transaction processing. We highlight some unique designs of G-Tran as follows:

- We design a *graph-native* data store with efficient data and memory layouts, which offers good data locality and fast data access for read/write graph transactions under frequent updates.
- We propose a *fully decentralized system architecture* by leveraging the benefits of RDMA to avoid the bottleneck from centralized transaction coordinating, and each worker executes distributed transactions under the MPP (i.e., massively parallel processing) model.
- G-Tran presents a *multi-version-based optimistic concurrency control* (*MV-OCC*) protocol, which is specifically designed to reduce the abort rate and CPU overheads in concurrent graph transaction processing.

We demonstrate the effectiveness of our system designs and the overall performance by comparing G-Tran with the SOTA graph databases [2, 4, 8, 26] using benchmark workloads [32, 44]. The

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 15, No. 11 ISSN 2150-8097.
doi:10.14778/3551793.3551813

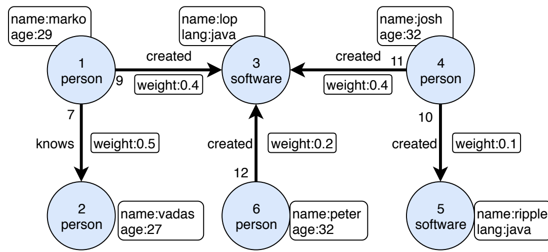


Figure 1: An example property graph

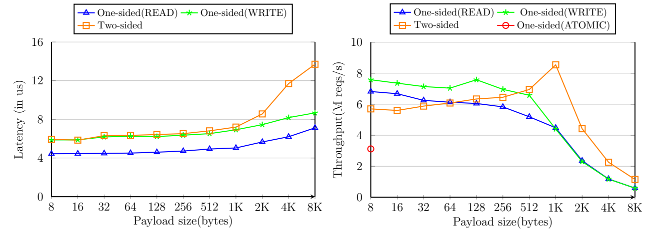


Figure 2: Latency and throughput of one-sided and two-sided RDMA primitives on different payloads

results show that G-Tran can achieve up to orders of magnitude improvements over the existing graph databases and obtain high throughput at both SR and SI isolation levels.

2 BACKGROUND

Property Graph. G-Tran adopts the *property graph* (*PG*) model [10] to represent graph data because of the expressiveness of PG. In a PG, vertices represent entities in an application and (directed) edges model the relationships between two entities. Both entities and relationships may have a set of properties to describe their attributes (e.g., names, gender, etc.) in the form of key-value pairs. We illustrate PG using an example in Figure 1. Each vertex/edge has a label to represent its role (e.g., *person*, *software*, *knows*, *created*) along with a set of properties, e.g., (*name*, "*marko*"), (*age*, *29*). Many existing graph databases, such as Titan [1], JanusGraph [2], OrientDB [9] and Neo4J [8], use PG to model their graph data.

Gremlin. Several graph query languages have been proposed including Gremlin [7], GQL [6], Cypher [5] and PGQL [60]. Among them, Gremlin has developed into a de facto standard for PG queries and supported by many graph databases due to its succinctness and expressiveness. Gremlin is a functional data-flow language proposed by Apache TinkerPop [3], which allows programmers to succinctly express a query as a sequence of *steps*. Each query step performs an atomic operation on vertices or edges. The details of Gremlin query steps can be found in [7]. G-Tran also uses Gremlin (the latest 3.0 standard) as its query language, and we summarize all Gremlin steps currently supported by G-Tran in Table 1.

Concurrency Control Protocols. Concurrency control ensures atomicity and isolation for database transactions. G-Tran supports both serializability (SR) and snapshot isolation (SI). SR has the strictest constraint that all concurrent transactions should execute their operations logically as if they are executed in sequence. SI relaxes the constraint to require that only all reads in a transaction should see a consistent *snapshot* of the database. There are three kinds of concurrency control protocols that are widely used in databases to implement different isolation levels: *two-phase locking* (*2PL*) [34], *optimistic concurrency control* (*OCC*) [42], and *multiple version concurrency control* (*MVCC*) [52]. 2PL is the most common and simplest protocol, which uses locks to avoid conflicts among concurrent transactions. OCC does not use lock, but it avoids conflicts by validation after a transaction completes its execution. Generally, OCC handles transactions in three steps: *process*, *validate*, and *commit/abort*. In comparison, MVCC provides *point-in-time consistent views* for multiple transactions at the same time by maintaining multi-versions of each object with timestamps, which incurs a higher storage overhead.

InfiniBand and RDMA. InfiniBand has become quite commonly in use and led to the development of many new distributed systems [33, 38, 41, 70]. InfiniBand offers two network communication stacks: IP over InfiniBand (IPoIB) and *remote direct memory access* (*RDMA*). IPoIB implements a TCP/IP stack over InfiniBand to allow current socket-based applications to be executable without modification. In contrast, RDMA provides a *verbs* API, which enables zero-copy data transmission through the *RNIC* without involving the OS. RDMA has two verb operations: two-sided *send*/*recv* verbs and one-sided *read*/*write* verbs. Two-sided verbs provide a socket-like message exchange mechanism, which still incurs CPU overheads on remote machines. One-sided verbs can directly bypass both the kernel and CPU of a remote machine to achieve low latency.

3 CHALLENGES AND DESIGN CHOICES

We have briefly discussed the characteristics of graph transaction processing and the limitations of existing graph databases in §1. In this section, we analyze the important factors that should be considered in the design of a high-performance distributed graph database. We summarize the challenges of distributed graph transaction processing as follows.

(C1) Graph data can easily result in poor locality for reads and writes in databases after continuous updates due to its irregularity [56].

(C2) Due to the power-law distribution on vertex degree and the small world phenomenon of most real-world graphs, the cost of traversal-based queries can be very high after multi-hops (e.g., ≥ 2) fan-out [54].

(C3) As a result of C2, graph transaction can involve larger read/write sets. In addition, the connectedness of graph data may lead to higher contention/abort rate and lower throughput in transaction processing. Thus, the transaction processing time also becomes longer compared with other transaction workloads such as in RDBMS and KVS.

(C4) Latency and scalability is another critical issue for distributed graph transactions. The network bandwidth, contention likelihood, and CPU overheads are the main bottlenecks [69]. Traditional centralized system architecture (i.e., assigning a master node as the global coordinator) may limit the scalability for OLTP.

We conduct experiments in §6 to verify the above issues. Here, we only discuss the design choices to address these challenges.

We first consider (C4). A significant overhead for distributed transaction processing is the large number of round-trip messages that are needed to ensure ACID. In recent years, many systems [12, 23, 29, 39, 49, 55, 62, 69] have been proposed to improve the performance of distributed transaction processing by leveraging RDMA

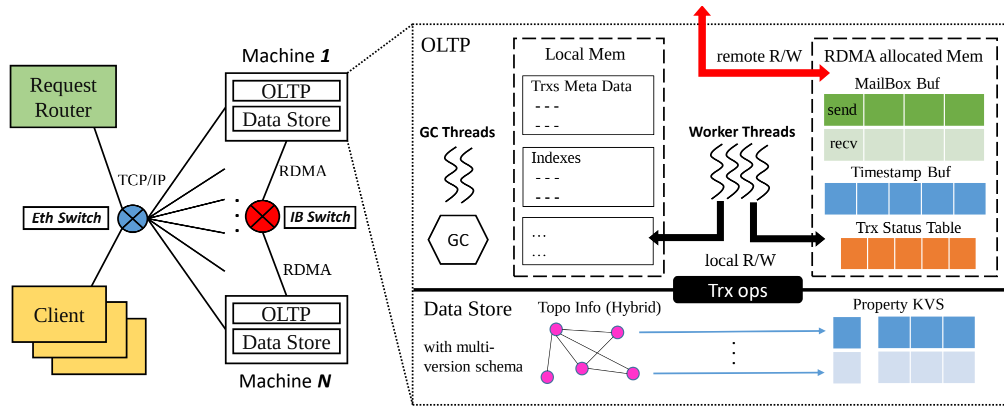


Figure 3: The architecture overview of G-Tran (best viewed in color)

as it can remove the overheads on network and CPU, although these works mainly focus on RDBMS or KVS instead of graph databases. Using RDMA for graph transaction processing can also improve the performance in terms of throughput, latency and scalability. However, the design of an RDMA-based distributed in-memory database is non-trivial. Specifically, the data store of G-Tran changes from a pure shared-nothing or shared memory architecture to a hybrid memory layout via RDMA connection. We leverage hybrid RDMA primitives (i.e., both one-sided and two-sided) in the OLTP layer to minimize the cost (i.e., network round trips and memory contention) of distributed transaction by combining the memory storage layout (described in §4). This requires us to take an overall redesign of the system from the storage layer to the OLTP layer in order to tightly integrate the system components. To better understand the performance behaviors of different RDMA primitives and to guide our design choices accordingly, we conducted an experimental analysis on the latency and throughput of one-sided and two-sided primitives as shown in Figure 2, using one machine as the client for sending and one machine as the server for receiving (machine configuration is given in §6). Generally, one-sided primitives (read and write) achieve lower latency and higher throughput than two-sided primitives. But when the message payload is larger than 1K, two-sided primitives can perform better in terms of throughput. Another observation is that, although one-sided atomic is relatively slower, it can still achieve 3.12M requests/sec on each machine, which is higher than the requirements of many workloads (e.g., LDBC). Thus, remote atomic will not be the bottleneck on the performance of transaction execution (as evaluated in the transaction workload in §6).

To address (C1) and (C2), we propose a new storage layout for property graphs to achieve both good data locality and efficient data access. It stores graph data under a graph-native schema, but organizes the (arbitrary-length) adjacency-lists of vertices into fixed-size rows and separately maintains vertex/edge properties into a key-value store. The storage layer of G-Tran splits the memory space of each machine into two parts. One part follows the shared-nothing architecture to store one piece of graph partition locally, and the other part follows the shared-memory architecture to compose an RDMA-based distributed memory space for remote property data access. Such design is based on the facts that: (1) implementing efficient RDMA-based data structures (e.g., map, tree) is non-trivial because the address space of each object should be recorded explicitly and this requires more memory footprint [75]; (2) not all data are necessary to be shared and to manage different regions of data in different memory spaces compactly can further improve the data access efficiency for local search and scan (§4.2).

For (C3), 2PL would incur high overhead due to locking on large portions of data. OCC is a good choice for read-intensive workloads, but we can further reduce the overhead of isolation maintenance in concurrent transactions by integrating OCC with multi-versioning. We adopt a *multi-version OCC* (*MVOCC*) protocol to coordinate concurrent transaction processing. MV-OCC has the following advantages. First, a multi-versioning mechanism allows read-only transactions to access old versions of objects without imposing any consistency overhead on read-write transactions. Second, OCC applies validation to check if there is any conflicting updates without locks. We propose our own optimistic optimizations in the MVOCC protocol to reduce the abort rate (§4.3).

Moreover, G-Tran adopts a decentralized architecture to process transaction execution (§4.4). Existing distributed databases [29, 31] follow a centralized approach to process tasks such as transactional metadata management and timestamp ordering on a master node. The master is likely a bottleneck of the entire cluster and affects the overall performance [69]. Instead, in our decentralized architecture, all servers share the load of each transaction according to the locality of its read/write sets and each server can handle multiple sub-queries simultaneously with multi-threads (i.e., MPP) for speed-up. In addition, RDMA can significantly reduce the costs of essential operations such as clock synchronization, timestamp ordering, transaction status synchronization in the decentralized architecture, which also helps improve the scalability.

4 SYSTEM DESIGN

4.1 Overview

Figure 3 shows the architecture of G-Tran. G-Tran supports multiple client connections via a regular Ethernet network. A request router is responsible for the assignment of incoming transactions from clients to the servers. G-Tran's server nodes are connected via InfiniBand, where each server has two layers, i.e., the storage layer and the OLTP layer.

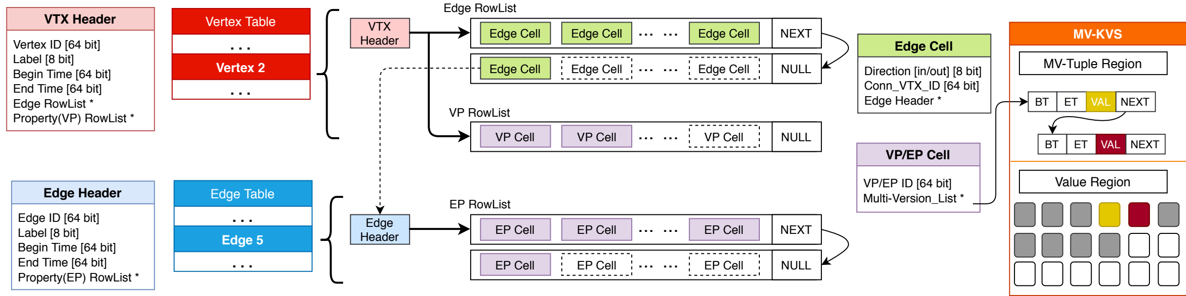


Figure 4: The data store of G-Tran (best viewed in color)

The storage layer keeps a property graph in two parts: *topology data* and *property data*. Topology data refer to the graph topology, i.e., vertices and edges in the format of adjacency lists. Property data are the keys and values of the vertex/edge properties. We partition a graph into N shards among N server nodes by an edge-cut partitioning strategy using hashing, where each shard stores one shard of vertices along with their in/out edges. G-Tran constructs a consistent global address space over all deployed nodes, so that the location of any object in the data store can be retrieved by its ID. The data store has a multi-version schema to support consistent view for concurrent transactions (§4.2).

The OLTP layer of each server node sets a group of worker threads to process incoming transactions. Worker threads interact with the data store to process data reads/writes. To allow remote data access and fast communication via RDMA, each server node registers a chunk of memory at NIC during the initialization stage, which divides the memory space of a server node into two regions, *RDMA allocated memory* and *local memory*, as depicted in Figure 3. We place different system components in different memory regions according to their functionalities, in order to enjoy the benefits of both local memory management (i.e., efficient maintenance and access of data structures such as tree, map, lock, etc.) and RDMA (i.e., low CPU overhead, fast remote data access and atomicity guarantee) for concurrent transaction processing. A cross-server transaction is executed simultaneously in the places where its data (i.e., the read/write sets) are located and each transactional operation can be processed in parallel based on its load. As a trade-off, MPP incurs more message round-trips for transaction coordination and consistency control, while we reduce such cost by leveraging RDMA with hybrid primitives (§4.4). We construct an RDMA-enabled mailbox (i.e., the green box in Figure 3) to process thread-level message sending/receiving, using one-sided and two-sided RDMA primitives. Besides the mailbox, in the RDMA allocated memory, G-Tran also maintains other system components (i.e., *timestamp bufs*, *transaction status table*) for global transactional metadata access, which is needed in the MV-OCC protocol (§4.3). For the transactions' private metadata (e.g., begin time, commit time, read/write set) and other structures (e.g., indexes), which do not need to be shared with other servers, are stored in the local memory.

4.2 Data Store

As shown in Figure 4, the data store in each server is composed of three components: *Vertex Table* (in red color), *Edge Table* (in blue color), and *Multi-Version Key-Value Store* (*MV-KVS*, in orange).

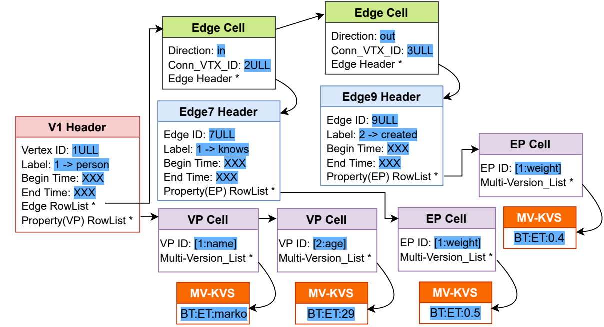


Figure 5: The example storage format of vertex 1 along with its edges 7&9 of Figure 1 in the data store

All vertex objects are stored in the Vertex Table. Every vertex object has a fixed-size *vertex header* to record the information (e.g., ID and label) of a vertex. The *begin time* and *end time* in the vertex header indicate the *visible time period* of the vertex, i.e., the period that the vertex is accessible to all active transactions. Usually, the begin/end time of a vertex is exactly the commit time of the transaction that creates/deletes this vertex. In addition, the vertex header also links to two *row-lists*, which record the connected edges (i.e., *Edge RowList **) and properties (i.e., *VP RowList **) of the vertex, respectively. To store the adjacency list of a vertex, we use an *Edge Cell* (in green) to represent each adjacent vertex, and then arrange all the Edge Cells in an adjacency list into rows in ascending order of the vertex IDs. Note that each row has a fixed number of cells. If one row is filled up, a new row will be allocated from the memory pool until the entire adjacency list is stored.

Each edge cell is mapped to one edge object stored in the Edge Table. Since an edge $e = (v_1, v_2)$ connects two vertices, there are two edge cells (of v_1 and v_2) pointing to the same edge object of e. If e is directed, then the edge cell of v_1/v_2 also keeps a direction sign to indicate that e is an out/in-edge of v_1/v_2. Each edge object also has an *edge header*, which records its ID, label, begin time, end time, and a link to the row-list, i.e., *EP RowList **, that stores the properties of the edge. Both VP RowList and EP RowList have the same layout as that of Edge RowList. Each *VP/EP Cell* in the VP/EP RowList records the ID of a property object and a pointer that links to its multi-version property values in the MV-KVS.

The MV-KVS is divided into two regions as shown in Figure 4: *MV-tuple region* and *value region*. The MV-tuple region stores a set of pre-allocated, fixed-size MV-tuples, where each MV-tuple records the begin and end time of a version of the corresponding property. All MV-tuples (i.e., the different versions) of a property

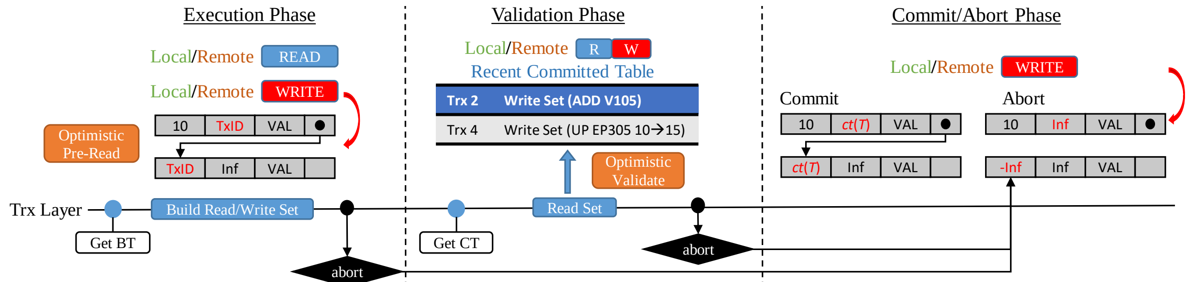


Figure 6: The MV-OCC protocol of G-Tran

object are ordered by their begin/end time for version searching during data access. Each MV-tuple keeps a pointer that links to where the value of this version is stored in the value region. We will discuss how to operate on the multi-versions of a property object in §4.3. Figure 5 uses vertex 1 (along with its edges) in the graph of Figure 1 as an example, to illustrate how G-Tran stores vertices, edges and their properties in the above data layout with the multi-version mechanism.

Design Principle. The key design of our data store is that all components have fixed sizes (except property values with variable lengths) and are aligned compactly in memory space wherever possible. This provides good data locality and is critical for efficient read and write (e.g., insert a new edge). As dynamic memory allocation is a costly operation, G-Tran uses memory pooling to pre-allocate memory buffers for all type of components (i.e., headers, cells, row-lists, tuples). This data layout enables that graph traversals will be executed as a combination of row-based scanning and object-based filtering (on MV-KVS) via *zero-CPU-overhead* one-sided RDMA read/write. In addition, it also benefits the multi-threaded execution of concurrent transactions (§4.4).

4.3 The MV-OCC Protocol

To guarantee a transaction T is serializable, we should hold two features: *Read stability* and *Phantom avoidance*. Usually, they can be implmented by locks or re-scan on the read-set, but obviously this approach has a high cost. We propose our own MV-OCC protocol with specific optimizations to avoid re-scan and coarse-grained locks. Our MV-OCC follows a general procedure of OCC but combines with our multi-version-based commit/abort rules. Before we discuss the details, we first define the basic components and some necessary concepts.

Transaction Status Table (TST) is a distributed table maintained in the RDMA-allocated memory of each server node. TST records the real-time status (i.e., *execution*, *validation*, *commit*, *abort*) of each *active transaction* T. Once T starts the processing of a new phase, its worker thread will update T's status in TST through RDMA atomic write. When another worker thread wants to check T's status for its own transaction processing, it can apply one-sided RDMA read to fetch the value with microseconds latency.

Recent Committed Table (RCT) is another distributed table but stored in the local memory of server nodes. RCT records the metadata of all *active read-write transactions* including their IDs, commit times and write sets, in order to enable relatively cheap and fine-grained validation mechanism. The read-set of a transaction T will also be recorded during the execution and we free it after the validation phase. The RCT is indexed by a B-Tree using the commit time of each T as the index key.

Multi-Versioning Read/Write Rules. Figure 6 shows the workflow of a transaction in G-Tran. When a transaction T reads a property object, it finds the visible version of the property value, i.e., the MV-tuple whose *begin time* (*BT*) and *end time* (*ET*) overlaps with the BT of T. If T wants to update the property object, it creates a new version, i.e., a new MV-tuple, and inserts its transaction ID (i.e., $TxID$) into the ET field of the current version and the BT field of the new version, as shown by the two MV-tuples in the Execution Phase in Figure 6. This indicates that the current property object is in the process of being updated, where the current version has been "locked" and the new version has not been committed yet. Then, if T commits successfully later on, these two fields will be replaced by the *commit time* (*CT*) of T, i.e., $ct(T)$ in the Commit/Abort Phase of Figure 6. This indicates that the current version has ended at $ct(T)$ and a new version beginning at $ct(T)$ is created. But if T aborts, then the BT field of the new version will be set as *-Inf* to indicate that this version has become invisible forever. All the old versions whose ET is before the earliest BT of all *active transactions*, as well as the versions with *-Inf* BT, will be cleaned and recycled back to the memory pool by garbage collection (§5).

We now describe the details of the specific three phases of the transaction workflow in Figure 6.

In the execution phase, a transaction T first obtains its BT, i.e., the timestamp when its processing starts, for version visibility checking. Then, T is executed while its read/write set is constructed by accessing the Vertex/Edge Tables and the MV-KVS in data store, following the multi-versioning read/write rules. We propose an *optimistic pre-read* mechanism in the execution phase to reduce the abort rate. We illustrate the idea by the example given in the Execution Phase in Figure 6, where a transaction T is doing the read-scanning on MV-tuples. Assume that T's BT > 10, then T will read the version (let it be V1) whose BT is 10 and ET is $TxID$ (meaning that another transaction $TxID$ is in the process of updating the corresponding property object). In this case, instead of aborting directly, optimistic pre-read assumes that $TxID$ will commit successfully later and executes T according to the status of $TxID$ as follows. There are four possible cases:

- $TxID$ is in *execution*: the version (let it be V2) next to V1 is a new version created by $TxID$ and V2 has not been committed yet. In this case, T reads V1 and validates the read-set consistency (i.e., a new version is indeed not committed) in its validation phase.

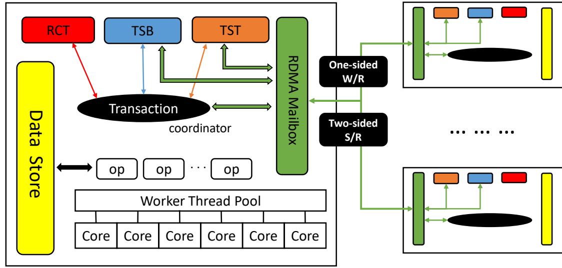


Figure 7: Distributed transaction processing in G-Tran

- $TxID$ is in *validation*: we optimistically assume that $TxID$ will commit, and thus T directly pre-reads V2 now but validates the commit dependency (i.e., if $TxID$ is indeed committed) in T's validation phase. Note that reading V1 causes read instability and leads to abort.
- $TxID$ is in *commit*: T directly reads V2 as $TxID$ has committed and the CT of $TxID$ is definitely earlier than the CT of T.
- $TxID$ is in *abort*: T ignores the new version V2, and reads the current version V1.

In the case of T is a read-write transaction, if T sees $TxID$ in the visible MV-tuples when T is inserting a new version, it should abort itself immediately (except $T = TxID$), since this case belongs to a write-write conflict.

In the validation phase, a read-write transaction T first obtains its CT, which is the timestamp when the validation begins (i.e., when the transaction logically commits). Then, T checks read stability and phantom avoidance through conflict checking. Specifically, based on the BT and CT of T, i.e., $bt(T)$ and $ct(T)$, we search from RCT for the active read-write transactions, *W-Trxs*, whose commit time falls in the period $[bt(T), ct(T)]$, because their write sets may change the read set of T. If no overlapping element exists between the read set of T and the write sets of *W-Trxs*, T can commit successfully. Otherwise, T should abort. The above conflict checking is executed as *two set comparison* on all server nodes simultaneously in a MPP manner (§4.4). We also propose an *optimistic validation* strategy to improve the success rate of commit. During the validation of T, if we find that T is in conflict with another transaction $TxID$, where $TxID$ is in the validation phase too, we do not abort $TxID$ immediately. Instead, we optimistically assume that $TxID$ will abort later and continue the validation process of T after recording such a dependency between T and $TxID$ (i.e., T should commit only if $TxID$ does abort).

At the end of the validation phase of T, we perform status checking for all the dependent transactions (if any). T can commit itself only if the following two conditions are met: (1) all its dependent transactions due to optimistic pre-read have committed, and (2) all its dependent transactions due to optimistic validation have aborted. Note that if T is a read-only transaction, its validation phase only needs to check those dependent transactions due to optimistic pre-read, because T has no write set.

In the commit/abort phase, a read/write transaction T physically effects its write set if T commits, or discards if T aborts, where the corresponding MV-tuples are updated accordingly based on the multi-versioning read/write rules.

Design Principle. Following our MV-OCC protocol, there is no coarse-grained read/write locks or re-scanning during the entire transaction processing, which improves both the throughput and latency of transaction processing. The rationality of adopting the two optimistic optimizations in the execution and validation phases is that graph transaction workloads are mostly read-heavy as mentioned in §3. Moreover, to achieve snapshot isolation, our protocol only needs to skip the optimistic pre-read in the execution phase and the entire validation phase, then to perform the regular commit (or abort when write-write conflict happens).

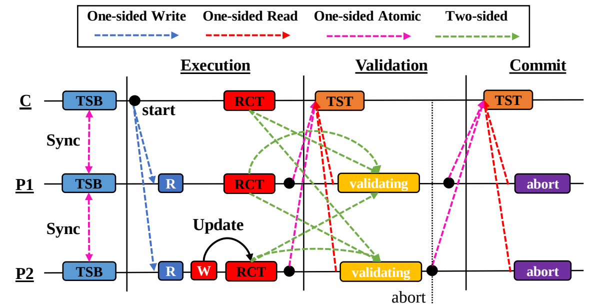


Figure 8: A phase-by-phase overview of transaction processing with MV-OCC (C and P stand for the coordinator and participants)

4.4 Distributed Transaction Processing

With the use of RDMA, the design goal of G-Tran's execution engine is to effectively parallelize transaction processing among servers while removing the computation bottlenecks (e.g., centralized coordinator, stragglers, locks). Existing RDMA-enabled databases [23, 62, 69] usually apply RDMA atomic primitives, i.e., compare-and-swap (CAS) and fetch-and-add (FAA), to lock and fetch records from remote machines to a local machine, and then perform the local transactional updates before writing them back to remote machines. However, such an approach is not efficient for graph transactions due to their large read/write sets. It is expensive to apply CAS & FAA and locks on large amounts of data at a time, and they will incur high CPU overheads and impair throughput.

Instead of simply fetching data remotely as in [23, 62], G-Tran uses different RDMA primitives in different execution phases to obtain the best performance gain. Guided by the performance of one-sided and two-sided primitives shown in Figure 2 in §3, we scale out the execution across servers through message passing by one-sided primitives to achieve low latency, and we use two-sided primitives in the validation phase as this phase has a large payload because of the large write/read sets of graph transactions. Atomic primitives are only considered on bit-level remote operations.

Figure 7 shows the decentralized architecture of G-Tran, where all servers have the same layout and play the same roles. Figure 8 illustrates how our protocol performs in each phase of transaction processing. Without loss of generality, we only discuss the processing of one particular transaction T as follows.

When a server receives T's request (assigned by the request router), it becomes the unique *coordinator* of T and takes charge of (1) transactional meta management, i.e., to update T's status in the local shard of *TST*, and (2) query management, i.e., to aggregate intermediate/final results if needed (e.g., for operators COUNT, MAX, etc.). Both TST and RCT are read/write-enabled, shared data structures via RDMA. In order to maintain the atomicity and consistency

of TST/RCT, we adopt different RDMA primitives accordingly based on their workloads. Each update in TST is executed by an RDMA atomic write to the coordinator in order to avoid unsafe updates from other servers. Applying atomic operation here is acceptable as a transaction status occupies only 2 bits. However, inserting an entire write set into RCT has a high payload and executing it on the RDMA allocated memory may raise lock contention and scalability issue. That is the reason why we maintain RCT in the local memory of each server, while using two-sided RDMA send/recv to query and update the RCT entries with read/write consistency guarantee. In addition, we also maintain a *timestamp buf (TSB)* in the RDMA allocated memory of each server to synchronize the earliest BT among the active transactions on all servers using one-sided atomic. Then, we can fetch the global earliest BT locally, which is used by RCT to garbage-collect the expired entries (i.e., transactions and their write sets).

The whole processing of transaction T is triggered by its coordinator via one-sided write and the processing load will be distributed to multiple servers based on the actual data locality of the read/write set of T (e.g., 2 participant servers are used in Figure 8). According to the protocol, when T needs to update its status, for example, a server has to abort T due to a local conflict, it will synchronize T's updated status to the shard of TST on the coordinator via one-sided atomic. Thus, at the beginning of each operation in T's processing, the first step G-Tran has to do is status checking. If T's status has been changed, then we need to switch the processing of T to the corresponding phase. In the above example, T will abort and terminate on all servers once they observe T's new status. Thus, TST works as a global flag for each active transaction, and all servers can fast access/update it through RDMA. In contrast, the read/write sets of T constructed in the execution phase of MV-OCC are recorded in each server locally. And in the validation phase of T, it first fetches all potential conflicting transactions from each shard of RCT, and then does conflict checking for their read/write sets on the servers locally as shown in Figure 8. Finally, if T can commit successfully, the coordinator will aggregate the partial query results from each participant and send the aggregated result back to the participants.

The MPP model helps speed up query processing and address the issue of skewed workloads. If one query step has a higher load, G-Tran will assign more threads to process this query step in parallel. As a high-performance system, we also take into consideration the side-effects brought from NUMA architecture [30, 50]. As shown in Figure 7, each thread in G-Tran's *worker thread pool* is bound with one CPU core to achieve better cache locality and memory locality on CPUs in cross-NUMA nodes. We omit the details here as this optimization is not the focus of our paper.

5 SYSTEM IMPLEMENTATION

G-Tran was implemented in C++, currently with 46K+ lines of code. We used *librdma* libaray to construct the mailbox for thread-level one-sided and two-sided RDMA primitives. In addition, we also provide a general TCP-based version, i.e., G-Tran without RDMA, which uses ZeroMQ TCP sockets to achieve point-to-point communication and uses MPI to coordinate the inter-process communication over Ethernet. Here we briefly discuss some implementation details of the system, while the source code and related tutorials are available in our codebase.

Table 1: The Gremlin Steps currently supported in G-Tran

Type	Query Steps
Init	g.V(), g.E()
Traversal	in, out, both, inE, outE, bothE, inV, outV, bothV
Update	addE, addV, property, drop
Filter	has, hasNot, hasKey, hasValue, hasLabel
GetValue	key, label, properties, values
Range	range, limit, skip, tail
Math	sum, max, min, mean, count
Aggregation	order, dedup, aggregate, group, groupCount
MapFilter	where, select, as, is, cap
BranchFilter	and, or, not, union, repeat

Gremlin Steps. G-Tran supports 48 Gremlin query steps currently based on the Gremlin v3.4.0 [7] standard. These steps are sufficient to be used for expressing a wide range of graph queries from real world. We list all of them in Table 1 and classify them into 10 types according to their functionalities.

Timestamp Ordering. Unified timestamp (together with MV-OCC) allows concurrent transactions to read a consistent snapshot of the database, while the timestamp order should match with the real time order. We follow the solution in [55] for global time synchronization using Marzullo's algorithm [48], where any server in the cluster can play the role of clock master and other servers periodically synchronize their clocks via RDMA writes.

Garbage Collection (GC). MVCC-based protocol usually has a high overhead on the storage. Thus, as an in-memory graph database, GC is critical for G-Tran to avoid the growth of memory consumption when servers run continuously. G-Tran's GC was implemented through one scanning thread and two GC threads as default, but users may configure the exact number of GC threads based on their workloads. However, recycling the memory slots occupied by obsolete objects requires write locks to guarantee data consistency for active read/write transactions, which leads to a degradation in transaction throughput and latency. In order to reduce the impact of GC, we separate the scanning process and execute GC jobs batch-by-batch. The scanning process collects the *obsolete objects*, which include all old versions whose visible time periods have expired, all invalid versions that are generated due to the aborted transactions, and all empty Edge/VP/EP rows that have been deleted. We pack different types of obsolete objects into different types of GC tasks based on their costs, where each type of tasks handles only one type of obsolete objects and a specific threshold is set to control the batch size. The scanning thread periodically scans and collects obsolete objects. Once a batch has been collected, the GC threads will be activated to garbage-collect these objects. We show the importance of GC by experiments in §6.4.1.

Indexes. Indexes are critical to achieve efficient query operations such as HAS and WHERE. G-Tran supports standard indexes (e.g., hashtables, B+ trees) on text and numerical values for fast look-up or range search on vertex/edge labels and property values. Users can specify the type of indexes to be constructed and the specific target keys (e.g., a certain property) for indexing via a client console. Then, G-Tran servers coordinate with each other to build a distributed index map in memory. The indexes can be updated accordingly once the related read-write transaction has committed, in order to keep the data consistent.

Table 2: Property graph datasets

Dataset	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert VP\rvert$	$\lvert EP\rvert$
LDBC-S	23,850,377	139,854,135	153,761,078	37,769,010
DBPedia	29,130,775	22,623,812	79,600,170	22,623,763
LDBC-L	81,585,767	495,119,129	441,220,072	142,182,014
Amazon	37,671,279	338,255,928	127,123,473	493,345,892

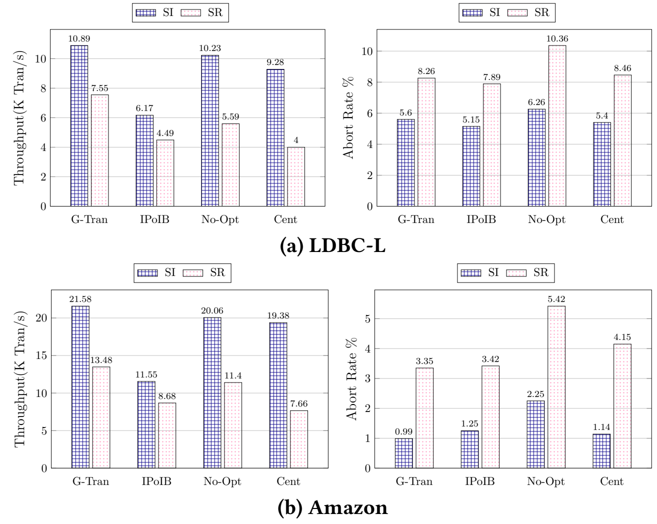
Fault Tolerance. Currently, G-Tran as a research prototype system, has not yet implemented fault tolerance. We plan to support fault tolerance as follows. We will use a standalone storage engine (e.g., RocksDB) to store all data entities (i.e., Headers, Rows and the multi-versioning value objects in the MV-KVS) in the data store. The original in-memory data store will be treated as a cache layer to cache all accessed entities for incoming queries. We can apply the LRU/LFU cache strategy to swap cache entities back from memory to disk when the memory consumption is high. For a read-write transaction T, we need to not only update the values in the cache but also generate a write-ahead log (WAL) as the redo log for each entity in T. Then, this WAL should be flushed to the persistent store with an *encoded key* for data durability. To follow the transactional protocol, we consider that a transaction T is successful only after all its WAL related operations are completed. When failure occurs, we load involved entities from the persistent store to the in-memory cache, and also fetch all WALs belonging to the entities (if any) using the *encoded keys*, and apply them to the entities for recovering the latest values.

6 EXPERIMENTAL EVALUATION

We compared G-Tran with four popular graph databases, JanusGraph v.0.3.0 [2], ArangoDB v.3.6.2 [4], Neo4J v.3.5.1 [8] and TigerGraph Developer Edition [26]. The experiments were run on a cluster of 10 machines, each equipped with two 8-core Intel Xeon E5-2620v4 2.1GHz processors, 128GB memory and Mellanox ConnectX-3 40Gbps Infiniband HCA, running on CentOS 6.9 with OFED 3.2 Infiniband driver. For fair comparison, we always used 20 computing threads in each machine (unless otherwise specified) for all the systems, and tuned the configurations of each system to give its best performance as we could. All query latency reported are the average of five runs and all throughput values are averaged over 300 seconds.

Datasets. We used four datasets: one small and one large synthetic property graphs created by LDBC-SNB[1] data generator, and two real-world property graphs crawled from DBpedia[2] (including two parts, citation data and citation links) and Amazon Product[3] (including product reviews and metadata). The small/large datasets were used for the system evaluation on the single-machine/distributed environment respectively. Table 2 lists the statistics of each dataset.

Query Benchmarks. Lissandrini et al. [44] proposed a benchmark (denoted as **LBV**) for graph database evaluation, which includes five categories of queries: *Creation(C)*, *Read(R)*, *Update(U)*, *Deletion(D)* and *Traversal(T)*. *R* and *T* belong to *READ* transactions, while *C*, *U* and *D* belong to *WRITE* transactions. We selected, with equal probability, most of the queries (i.e., [C:Q2-Q7], [R:Q11-Q15], [U:Q16-Q17], [D:Q18-Q21], [T:Q22-Q27]) in the benchmark (except few that are not for transactions, e.g., Q8-Q9 for counting all



(a) LDBC-L

(b) Amazon

Figure 9: The effects of individual system design

V/E) for throughput evaluation. Also, we selected 8 heavy queries (i.e., IC1-IC4 and IS1-IS4) from **LDBC SNB** benchmark [32] for single-query latency evaluation. Note that we did not use the LDBC benchmark for comprehensive evaluation as it does not support UPDATE/DELETE workloads and the queries are only applicable on its own synthetic datasets. We list the templates of above benchmark queries in the wiki page [4] of our codebase, where the specific query values (e.g., VID, VPIDs, EPIDs, etc.) in the query templates were randomly sampled from the respective datasets.

6.1 Evaluation of System Designs

We first evaluate the effectiveness of the various system designs, including the data store, the decentralized architecture, the optimizations in MV-OCC (i.e., optimistic pre-read and optimistic validation), and the speed-up due to RDMA-related designs.

6.1.1 ***Evaluation of Individual System Designs.*** To examine the effect of each individual design on the system performance, we created three variants of G-Tran: G-Tran without RDMA but using IPoIB (denoted as **IPoIB**), G-Tran without the two optimizations in MV-OCC (denoted as **No-Opt**), and a centralized version (denoted as **Cent**) with a global master (i.e., transaction coordinator). We tested them on two large graphs, LDBC-L and Amazon, using 8 machines. We used a mixed workload formed by READ and WRITE queries in the LBV benchmark. Half of the queries are WRITE, which create a relatively high-contention scenario.

Figure 9 reports the transaction throughput and the abort rate of G-Tran and its three variants, at both SI and SR isolation levels. Compared with G-Tran, the throughput of its IPoIB variant is reduced around 30% - 50%, which is due to the higher latency of normal network connection and the extra CPU overhead between NIC and the OS kernel. However, the IPoIB variant still significantly outperforms existing systems as reported in §6.2, which shows that other system designs also play important roles in G-Tran's high performance.

[1] http://ldbcouncil.org/developer/snb
[2] https://wiki.dbpedia.org/downloads-2016-10
[3] http://jmcauley.ucsd.edu/data/amazon/
[4] https://github.com/yaobaiwei/GTran/wiki

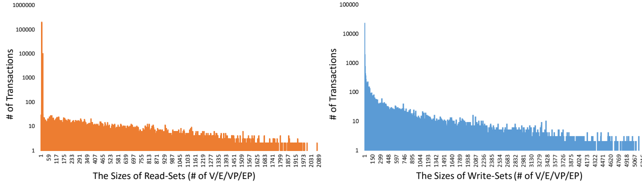


Figure 10: The distribution of the sizes of the read/write-sets of the LBV graph transactions on Amazon

Table 3: The latency (in *msec*) of the LDBC queries

LDBC-S	IC1	IC2	IC3	IC4	IS1	IS2	IS3	IS4
G-Tran	7,085	189	4,986	347	0.5	13.1	4.6	0.4
Neo4J	8,962	824	9.6E4	1,249	1.1	25.4	6.8	1.6
J.G.	1.9E5	1.4E4	1.3E6	1.3E5	1.2	20.6	2.7	0.9
ArangoDB	1.4E5	1,420	9.1E4	3,149	1.1	58.9	33.6	0.8
T.G.(install + run)	4.5E4 +63.5	4.1E4 +19.1	4.4E4 +370	4.5E4 +21.3	3.8E4 +8.2	3.9E4 +15.2	3.6E4 +8.9	3.5E4 +6.7

Disabling optimistic pre-read and optimistic validation in the MV-OCC protocol also leads to a degradation of the throughput, especially at SR isolation. As shown in Figure 9(a), on the LDBC-L dataset, No-Opt's throughput (5.59K tps) at SR drops 25.9% compared with G-Tran's throughput (7.55K tps). No-Opt's throughput at SI is only slightly dropped as our multi-version-based solution can already eliminate much of the contention at SI isolation. However, the abort rate of the No-Opt variant is significantly higher. The increase in the abort rate is more obvious on the Amazon dataset as it is a real-world graph with a power-law distribution on vertex degree, leading to higher contention and abort rate. The result shows that our optimizations in the MV-OCC protocol can effectively improve the processing of concurrent graph transactions, which have large read/write-sets. We report the distribution of the sizes of the read/write-sets of the transactions in Figure 10, showing that although more transactions have relatively smaller read/write-sets, there are also a large number of transactions having large read/write-sets. This also explains the relatively high abort rate of G-Tran at SR compared with that at SI.

To demonstrate the advantage of the decentralized architecture over RDMA for distributed transaction processing, we further compare G-Tran with its Cent version. At SR isolation, Cent has the lowest throughput for both datasets (i.e, 4K tps and 7.66K tps). Because in Cent's setting, the master plays the role of a global coordinator which handles the coordinating tasks of all concurrent transactions. These tasks create significant CPU and network overheads on a single node, which becomes the bottleneck and limits the general processing power of the entire distributed system. We also observe an increase in the abort rate of the Cent version, because the averagely higher latency per transaction leads to higher contention and in turn increases the overall abort rate.

6.1.2 ***Evaluation of Data Store.*** Next, we evaluate how the design of the data store effects G-Tran's performance. Since we cannot disable the data store in G-Tran individually as we did for the other features, we conducted the experiments on a single-machine setting to exclude the influence of G-Tran's RDMA-aware components and decentralized architecture. In addition, before we started to run the workloads, we first warmed up each system by running the mixed LBV workload in §6.1.1 for an extended period of time, to simulate the real scenario that graph data locality has been broken after continuous updates. It leads to more random access on the entities of the graph, which can thus be used to indicate the effectiveness of the data layout.

Table 4: The latency (in *msec*) of k-hop traversal queries

DBpedia	Q1	Q2	Q3	Q4
G-Tran	0.9	9.7	966	8,084
Neo4J	1.8	20.5	1,128	19,217
ArangoDB	24.4	93.5	17,659	287,012
LDBC-S	**Q1**	**Q2**	**Q3**	**Q4**
G-Tran	0.3	349	5,338	42,452
Neo4J	1.4	758	9,911	128,762
J.G.	1.3	661	42,916	1,211,714
ArangoDB	0.6	2,006	36,715	> 4h
T.G.(install + run)	40,518 +8.82	59,968 +389	92,770 +3,788	132,766 +62,053

We used the 8 heavy queries mentioned above to evaluate the performance. Consider that the LDBC benchmark queries can only function on the LDBC synthetic dataset, we also involved a typical multi-hop query template into the evaluation with the format:

$$g.V().has([primary_key]).(both())^k \quad (1)$$

The *both()* operator returns both the in-neighbors and out-neighbors of a source vertex. Here, *both()* repeats $k = 1, 2, 3, 4$ times to represent a k-hop traversal from a starting vertex located by *has()* operator with a given primary key (e.g., name).

We compared G-Tran with JanusGraph, ArangoDB, Neo4J and TigerGraph, where all the systems ran on a single machine. Table 3 reports the latency of the LDBC benchmark queries on the LDBC-S dataset [5], and Table 4 reports the latency of the k-hop traversal queries on both the LDBC-S and DBpedia datasets. G-Tran achieves the shortest latency on almost all LDBC queries (except compared with JanusGraph on IS3). In particular, for the complex queries, i.e., IC1-IC4, G-Tran's latency is two orders of magnitude smaller than that of JanusGraph and ArangoDB. The gap between G-Tran and Neo4J is smaller but also 2-3 times in most cases. TigerGraph achieves competitive performance in its "run" stage, but every query needs an "install" stage before running and this "install" stage is costly. For the k-hop traversal queries, Table 4 shows the query latency on all systems increases exponentially as k increases, because the size of the read-set grows exponentially for each hop of traversal. But after traversing more than 2 hops, G-Tran starts to show orders of magnitude advantage over the other systems.

We explain the results by analyzing the storage design of each system as follows. Neo4J stores all edges of the entire graph into a global sequential table. Consequently, a graph traversal will suffer from jump addressing in physical storage space on Neo4J, because each newly inserted edge is appended directly to the tail of the sequential table without locality guarantee. As for TigerGraph, although it is not open source, considering that it requires users to indicate the edge type (e.g., friendship) during a traversal, we conjecture TigerGraph stores edges separately based on their types. Then, this layout will require an extra aggregation to merge those separate edge sets when a traversal involves multi-type edges, which explains why TigerGraph has poor performance on the k-hop traversal queries. Both JanusGraph and ArangoDB store edges

[5] JanusGraph failed to load DBPedia in 24 hours, and TigerGraph could not load DBPedia as it requires a fixed schema input while DBPedia has no schema.

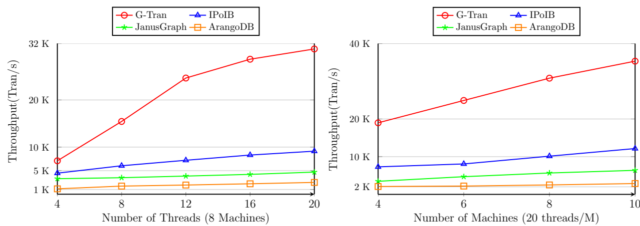


Figure 11: Scale-up and scale-out throughput on Read-Only workload over the LDBC-L dataset

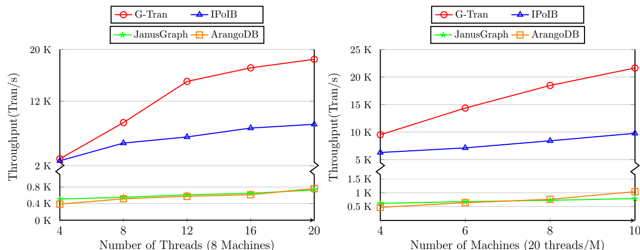


Figure 12: Scale-up and scale-out throughput on Read-Intensive workload over the LDBC-L dataset

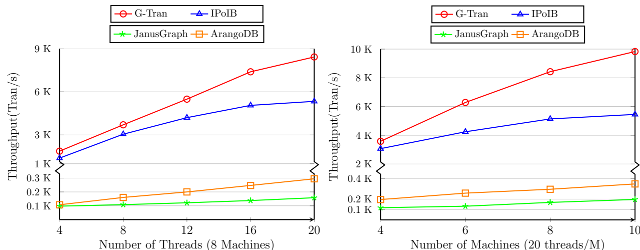


Figure 13: Scale-up and scale-out throughput on Write-Intensive workload over the LDBC-L dataset

as individual cells or collections. Specifically, JanusGraph stores edge cells of one vertex together with its property cells in one row, which incurs extra overhead for locating edges from properties. ArangoDB stores edges as collections of documents. Thus, the execution of multi-hop traversal needs to locate the connected edges of each vertex by searching on edge collections. In comparison, G-Tran divides arbitrary-length adjacency-lists into fixed-size rows, which provides more efficient edge insertion and deletion through allocating new rows or compacting under filled rows. Note that we allocate a new row only after filling up all blanks in the existing row and the sparse row compaction is processed periodically. As a result, traversals in G-Tran can access the edges of a vertex sequentially and the data locality will not be broken even after frequent updates.

6.2 Throughput Analysis

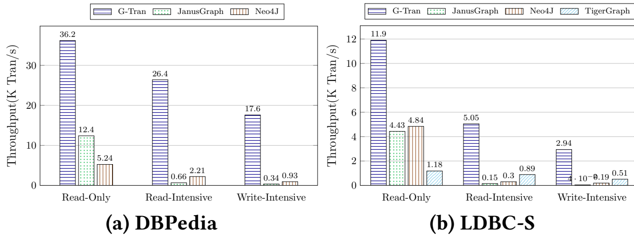
Now, we compared G-Tran with other systems for throughput analysis in both distributed and single-machine settings. We applied three workloads for evaluation: (1) *Read-Only* (*RO*), formed by READ queries in the LBV benchmark, (2) *Read-Intensive* (*RI*), which consists of 80% READ and 20% WRITE queries, (3) *Write-Intensive* (*WI*), which consists of 20% READ and 80% WRITE queries.

6.2.1 ***Distributed Processing.*** We first compared G-Tran and its IPoIB variant (G-Tran-IPoIB) with JanusGraph (using HBase as the storage backend) and ArangoDB. For Neo4J and TigerGraph, we only can download their Developer Edition which does not support distributed processing. We set isolation at SI because both ArangoDB and JanusGraph do not support SR.

Figures 11, 12, 13 report the throughput scalability of each system on the three workloads respectively over the LDBC-L dataset. Specifically, for scale-up evaluation, we depolyed all systems on 8 machines but varying the number of threads per machine from 4 to 20. For scale-out evaluation, we ran them on 4 to 10 machines with 20 threads/machine. G-Tran achieves significantly higher throughput than other systems over all three workloads. Although the performance of G-Tran-IPoIB is degraded when RDMA features are disabled, it still outperforms both JanusGraph and ArangoDB in all cases. This indicates that the use of RDMA in G-Tran is not the only reason for its high performance, but other system components are also important as discussed in §6.1. As we increase the number of machines from 4 to 10, the results show that using RDMA indeed brings an advantage to distributed processing because G-Tran has higher rate of increase in throughput than G-Tran-IPoIB. Overall, when more machines are used, G-Tran's throughput increases more on RI and WI workloads. This phenomenon can also be observed in scale-up performance. In contrast, the throughput of both JanusGraph and ArangoDB are relatively low and scales poorly. Besides caused by their data storage as we discussed in §6.1.2, this is also related to how the execution engine of each system processes the concurrent transactions. G-Tran's MPP model enables higher parallelism to process each transaction when more resources are available. Thus, these transactions can commit earlier and there is less contention when accessing data. However, both JanusGraph and ArangoDB follow the one-thread-one-transaction mechanism. When more threads are available, more transactions will join in the system and being processed simultaneously. This increases the contention among all concurrent transactions and incurs more overheads. Moreover, when more machines are involved, the graph will be partitioned into more shards, which breaks the locality of the graph and leads to extra overhead on communication.

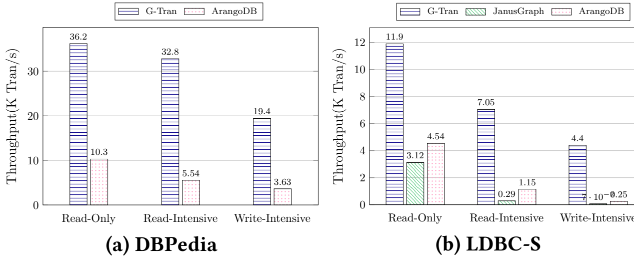
6.2.2 ***Single-Machine Processing.*** We compared G-Tran with 4 other systems for transaction throughput evaluation on single-machine at both SI and SR. We used BerkeleyDB as the backend of JanusGraph for its evaluation at SR, as only BerkeleyDB (a standalone engine) supports SR. Neo4J and TigerGraph do not support SI, while ArangoDB does not support SR. TigerGraph and JanusGraph (using HBase at SI) failed to load DBPedia as explained in §6.1.2.

As reported in Figure 14 and Figure 15, G-Tran achieves significantly higher throughput than other systems on all workloads at both SR and SI. Note that in single-machine setting, the performance advantages of G-Tran do not come from RDMA and its decentralized architecture, but mainly from its data store design and MV-OCC protocol. JanusGraph has worse performance than others because G-Tran, TigerGraph and Neo4J have native graph stores with tailored designs for transaction processing, while JanusGraph is built upon a general NoSQL-based store. G-Tran's good performance comes mainly from (1) its multi-version based storage, which enables lock-free snapshot reads; (2) the optimizations in MV-OCC protocol, which reduces contention among the concurrent transactions with less abort and retry; (3) its MPP execution engine, which enables parallel processing inside each transaction. By



(a) DBPedia (b) LDBC-S

Figure 14: Single-machine throughput at SR



(a) DBPedia (b) LDBC-S

Figure 15: Single-machine throughput at SI

Table 5: Memory footprint (GB) of each system at SI & SR

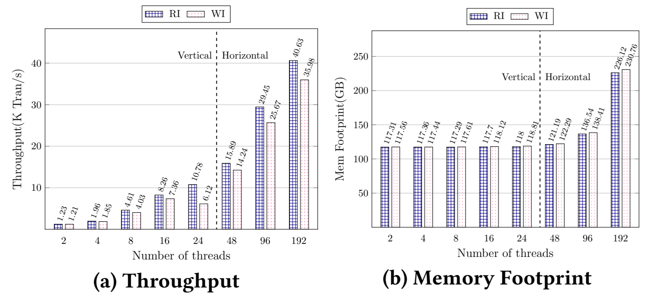
Workload		G-Tran	Neo4J	J.G.	ArangoDB	T.G.
SR	RO	117.30	3.44	4.46	-	6.34
	RI	117.53	7.35	22.32	-	6.47
	WI	117.92	11.46	24.25	-	6.60
SI	RO	117.84	-	2.43	4.24	-
	RI	117.89	-	20.1	4.98	-
	WI	117.96	-	22.62	11.09	-

contrast, Neo4J's transaction engine does not support SR natively but requires explicit locks on query language level to achieve SR. TigerGraph's low throughput is related to its costly "install" stage, which is used to pre-translate and optimize queries before their real execution. G-Tran requires neither explicit lock nor "install" stage to process transactions. In addition, ArangoDB can achieve higher throughput than JanusGraph due to its MVCC-based storage. But it has inefficient data layout on graph store, thus ArangoDB's throughput is still lower than that of G-Tran.

Memory Footprint. We also measured the memory footprint of G-Tran compared with the other systems for processing three workloads at SR and SI on LDBC-S. Table 5 shows that G-Tran uses significantly more memory than the other systems. However, we note that G-Tran uses the memory pooling technique to pre-allocate memory at one time during system initialization (mentioned in §4.2), which is why the memory usage of G-Tran is almost unchanged for different workloads, while the memory consumption of other systems increases as the workload increases. Specifically, G-Tran allocates 6.5GB/3.8GB/2.4GB for Edge/VP/EP RowList, 3.2GB/40.8GB for Vertex/Edge Table, 39.8GB/9.8GB for the VP/EP MVCC-Pool, 120MB for Mailbox, respectively, while the remaining memory usage is for runtime allocation (e.g., index, RCT, etc.).

6.3 Stress Test

To further analyze the system bottleneck at runtime, we conducted a stress test for G-Tran on both RI and WI workloads using the LDBC-S dataset, as these two workloads are more intensive. We first increased the number of threads in a single server from 2 to 24 (i.e., vertical scaling), and then we increased the number of servers



(a) Throughput (b) Memory Footprint

Figure 16: Scalability performance under stress test.

from 1 to 8 (i.e., horizontal scaling). For each setting, we issued an abundant number of transactions from multiple clients to the G-Tran servers until the whole system becomes saturated. Figure 16 reports the throughput and memory footprint of G-Tran under different settings. When the system is under stress, we observe the following bottlenecks. First, the throughput starts to drop on WI workload when the system expands from 16 cores to 24 cores. When we increase the number of cores to 24, the number of transactions issued also increases to saturate the system. However, processing too many transactions concurrently will result in higher abort rate and lower throughput because there are more write-write conflicts. The WI workload also has higher overhead and each transaction needs more time to be processed on average, which leads to more serious thread contention. Second, G-Tran shows good horizontal scalability for both RI and WI workloads, except when the number of servers is increased to 8. This is because the quality of graph partitioning decreases rapidly when more machines are used and hence the network communication cost also increases rapidly. Although RDMA significantly reduces the network communication cost, its effectiveness also decreases when the quality of graph partitioning is worsen. On the other hand, no increase in memory footprint is observed when the system scales vertically, while the memory footprint increases sub-linearly in the case of horizontal scaling. Thus, the result suggests that the in-memory data store is not a bottleneck.

6.4 Effects of Others

6.4.1 ***Garbage Collection***. We evaluated the effectiveness of Garbage Collection (GC) by enabling and disabling it during transaction processing. We conducted the experiment using 8 machines on the LDBC-L dataset and executed the same workload at SR isolation as we did in §6.1.1. Figure 17 and Figure 18 reports the real-time throughput and memory utilization of G-Tran for a period of 600 seconds from the beginning, i.e., as soon as G-Tran finishes the data loading. The throughput is higher at the beginning as there is not many GC jobs to do, and the system becomes stabilized at around 200 seconds. The result shows that, when GC is disabled, the memory consumption of G-Tran increases linearly with time. But when GC is enabled, the memory consumption remains relatively stable. We observe some obvious drops at 200s, 295s, 380s and 500s in Figure 18, which are due to the periodical garbage collection that releases the occupied memory and returns it back to the memory pool. During the whole process, the negative side-effects of GC execution on transaction throughput is minor. We can compare

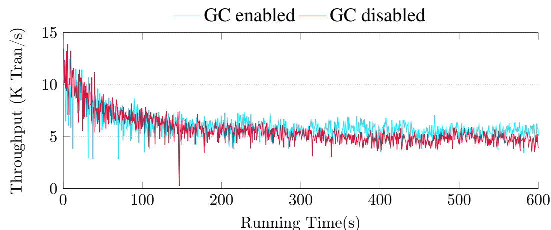


Figure 17: Throughput with GC enabled/disabled

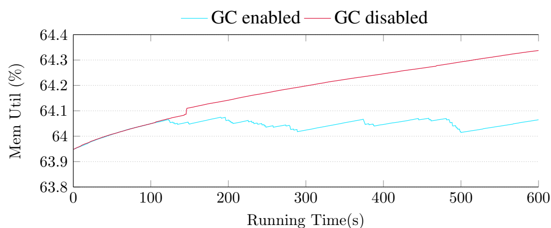


Figure 18: Memory consumption with GC enabled/disabled

Table 6: Index construction time and query latency (in msec)

	IC1	IC2	IC3	IC4	IS1	IS2	IS3	IS4
w/o index	22814	21699	26935	34895	22883	25287	22524	23938
w/ index	1559.6	69.27	2553.9	14518	1.54	4.63	2.22	0.724
index build	+134327 (msec)							
index mem	+5.39GB /machine avg.							

the two curves in Figure 17, the throughput of G-Tran in the GC-enabled case follows very closely to the case when GC is disabled, showing that our GC mechanism has low overhead. Actually, at the later period, in the GC-enabled case, G-Tran's throughput even has a tiny increase. This is because at this time, the system storage has been accumulated by many invalid/expired versions of various objects, to clean them up timely can de-fragment those sparse rows in the vertex/edge tables and accordingly improve the entire memory locality, which helps improve the efficiency of data scan and search. We can also observe a significant drop on the throughput in the GC-disabled curve at round 140s in Figure 17. While at the same time point, the GC-disabled curve for memory utilization in Figure 18 shows a non-continuous increase. We guess that at this moment, there were a large number of write transactions coming into the system, and system itself suffered from their processing and commits/aborts in short-term.

6.4.2 ***Index***. Finally, we analyzed the effects of index for query acceleration and also measured the overhead of index construction in terms of time and memory. We conducted the experiment on the LDBC benchmark queries using 8 machines to build index on the vertex ID for LDBC-L. As reported in Table 6, index can significantly reduce query latency with a relatively low cost, as index building takes only around 2 minutes and 5.4GB memory. Index works better for simple queries (i.e., IS1-IS4) because the complex queries (i.e., IC1-IC4) have more traversal steps (e.g., or(), union(), etc.) and more computation logics (e.g., order(), groupCount(), etc.), which cannot be accelerated by index only.

7 RELATED WORK

Graph Databases. Existing graph databases as we have discussed above like Neo4J [8], Titan [1], JanusGraph [2], ArangoDB [4] and OrientDB [9], have no modern MPP-based system architecture for distributed transaction processing and most of them support only low isolation level (e.g., *snapshot*, *read committed*). Both Grasper [18, 21] and TigerGraph [26] target at massive graph queries (i.e., MPP) with native graph store, but Grasper focuses only on OLAP workload instead of OLTP and TigerGraph is not designed for high performance. LiveGraph [74] is a single-machine graph database, which proposes a graph-aware data structure, named Transactional Edge Log (TEL), to enable purely sequential scans over the adjacency lists. A1 [15] is also an RDMA-based in-memory graph database built upon FaRM [28, 29], which applies Opacity and Multiversioning to achieve serializable transactions. The underlying data layout of A1 is a key-value store, which directly uses the FaRM's store. We cannot compare with A1 as it is not open source. The granularity of multi-versioning in G-Tran's data layout is a Cell, no vertex object or data object with a new version are created if we insert/delete an edge or update a property value. Such a layout design makes G-Tran more efficient for updates and data storage (i.e., requiring less memory space to maintain multi-versions).

Graph Processing Systems. Many graph processing systems have been proposed [11, 22, 36, 37, 53, 63, 66, 68, 72, 73] based on Pregel model [45] or other computation models [58, 67, 71]. But they focus on offline graph workloads such as PageRank, Connected Component and SSSP. There are also other systems that aim at complex graph analytics and mining [19, 20, 64, 65], or streaming graph processing [17, 46, 47, 61]. The system designs required for batch or streaming graph processing are quite different from those of a graph transaction database, which needs to address the side effects brought from transactional issues (e.g., locks, timestamps, validation, commit/abort, etc.), index construction, garbage collection and others for processing OLTP workloads.

Distributed Transaction Processing. Also, many general transaction processing systems have been proposed in recent, e.g., Google's Spanner [24], Granola [25], FaSST [40] and others [27, 51, 57, 59]. Some of them leverage new hardwares such as RDMA, HTM and NVM to achieve high performance (e.g., FaRM [28, 29] and DrTM [23, 62]). FaRM proposed an RDMA-friendly protocol to enable strict serializiable transactions with high throughput, low latency, and high availability. DrTM proposed an OCC protocol combining both HTM [14] and RDMA to ensure the strong consistency and atomicity. However, these systems are not specially designed for graph, which has its own unique challenges (§1 and §3).

8 CONCLUSION

We presented G-Tran, a high-performance distributed graph database built upon the decentralized architecture. To tackle the unique challenges of graph transaction processing, we used RDMA one-sided/two-sided primitives respectively in different system components to reduce system overheads from network and CPUs, combining with a graph-native data store and an optimized MV-OCC transaction protocol. G-Tran achieved an overall good performance in terms of both latency and throughput.

ACKNOWLEDGMENTS

We thank the reviewers for their constructive comments and suggestions that have helped improve the quality of the paper.

REFERENCES

[1] 2015. *TITAN.* http://titan.thinkaurelius.com/.

[2] 2019. *JanusGraph.* http://janusgraph.org/.

[3] 2022. *Apache TinkerPop.* http://tinkerpop.apache.org/.

[4] 2022. *ArangoDB.* https://www.arangodb.com/.

[5] 2022. *Cypher - the Neo4j query Language.* http://www.neo4j.org/learn/cypher.

[6] 2022. *GQL.* https://www.gqlstandards.org/.

[7] 2022. *Gremlin.* https://tinkerpop.apache.org/gremlin.html.

[8] 2022. *Neo4J.* https://neo4j.com/.

[9] 2022. *OrientDB.* https://orientdb.com/.

[10] Renzo Angles. 2018. The Property Graph Database Model. In *Proceedings of the 12th Alberto Mendelzon International Workshop on Foundations of Data Management, Cali, Colombia, May 21-25, 2018 (CEUR Workshop Proceedings)*, Dan Olteanu and Barbara Poblete (Eds.), Vol. 2100. CEUR-WS.org. http://ceur-ws.org/Vol-2100/paper26.pdf

[11] Ching Avery. 2011. Giraph: Large-scale graph processing infrastructure on hadoop. *Proceedings of the Hadoop Summit. Santa Clara* 11 (2011).

[12] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. 2016. The End of Slow Networks: It's Time for a Redesign. *PVLDB* 9, 7 (2016), 528–539. https://doi.org/10.14778/2904483.2904485

[13] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry C. Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkateshwaran Venkataramani. 2013. TAO: Facebook's Distributed Data Store for the Social Graph. In *2013 USENIX Annual Technical Conference, San Jose, CA, USA, June 26-28, 2013.* 49–60. https://www.usenix.org/conference/atc13/technical-sessions/presentation/bronson

[14] Trevor Brown and Hillel Avni. 2016. PHyTM: Persistent Hybrid Transactional Memory. *PVLDB* 10, 4 (2016), 409–420. https://doi.org/10.14778/3025111.3025122

[15] Chiranjeeb Buragohain, Knut Magne Risvik, Paul Brett, Miguel Castro, Wonhee Cho, Joshua Cowhig, Nikolas Gloy, Karthik Kalyanaraman, Richendra Khanna, John Pao, Matthew Renzelmann, Alex Shamis, Timothy Tan, and Shuheng Zheng. 2020. A1: A Distributed In-Memory Graph Database. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 329–344. https://doi.org/10.1145/3318464.3386135

[16] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. 2006. Bigtable: A Distributed Storage System for Structured Data (Awarded Best Paper!). In *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*, Brian N. Bershad and Jeffrey C. Mogul (Eds.). USENIX Association, 205–218. http://www.usenix.org/events/osdi06/tech/chang.html

[17] Cheng Chen, Hejun Wu, Dyce Jing Zhao, Da Yan, and James Cheng. 2016. SGraph: A Distributed Streaming System for Processing Big Graphs. In *Big Data Computing and Communications - Second International Conference, BigCom 2016, Shenyang, China, July 29-31, 2016. Proceedings (Lecture Notes in Computer Science)*, Yu Wang, Ge Yu, Yanyong Zhang, Zhu Han, and Guoren Wang (Eds.), Vol. 9784. Springer, 285–294. https://doi.org/10.1007/978-3-319-42553-5_24

[18] Hongzhi Chen, Changji Li, Juncheng Fang, Chenghuan Huang, James Cheng, Jian Zhang, Yifan Hou, and Xiao Yan. 2019. Grasper: A High Performance Distributed System for OLAP on Property Graphs. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2019, Santa Cruz, CA, USA, November 20-23, 2019.* ACM, 87–100. https://doi.org/10.1145/3357223.3362715

[19] Hongzhi Chen, Miao Liu, Yunjian Zhao, Xiao Yan, Da Yan, and James Cheng. 2018. G-Miner: an efficient task-oriented graph mining system. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018.* 32:1–32:12. https://doi.org/10.1145/3190508.3190545

[20] Hongzhi Chen, Xiaoxi Wang, Chenghuan Huang, Juncheng Fang, Yifan Hou, Changji Li, and James Cheng. 2019. Large Scale Graph Mining with G-Miner. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019.* 1881–1884. https://doi.org/10.1145/3299869.3320219

[21] Hongzhi Chen, Bowen Wu, Shiyuan Deng, Chenghuan Huang, Changji Li, Yichao Li, and James Cheng. 2020. High Performance Distributed OLAP on Property Graphs with Grasper. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 2705–2708. https://doi.org/10.1145/3318464.3384685

[22] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. 2015. PowerLyra: differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems, EuroSys 2015, Bordeaux, France, April 21-24, 2015.* 1:1–1:15. https://doi.org/10.1145/2741948.2741970

[23] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. 2016. Fast and general distributed transactions using RDMA and HTM. In *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016, London, United Kingdom, April 18-21, 2016.* 26:1–26:17. https://doi.org/10.1145/2901318.2901349

[24] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2013. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst.* 31, 3 (2013), 8:1–8:22. https://dl.acm.org/citation.cfm?id=2491245

[25] James A. Cowling and Barbara Liskov. 2012. Granola: Low-Overhead Distributed Transaction Coordination. In *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012.* 223–235. https://www.usenix.org/conference/atc12/technical-sessions/presentation/cowling

[26] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor Lee. 2019. TigerGraph: A Native MPP Graph Database. *CoRR* abs/1901.08248 (2019). arXiv:1901.08248 http://arxiv.org/abs/1901.08248

[27] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Åke Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL server's memory-optimized OLTP engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013.* 1243–1254. https://doi.org/10.1145/2463676.2463710

[28] Aleksandar Dragojevic, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014.* 401–414. https://www.usenix.org/conference/nsdi14/technical-sessions/dragojevi%C4%87

[29] Aleksandar Dragojevic, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. No compromises: distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015.* 54–70. https://doi.org/10.1145/2815400.2815425

[30] Andi Drebes, Antoniu Pop, Karine Heydemann, Nathalie Drach, and Albert Cohen. 2016. NUMA-aware scheduling and memory allocation for data-flow task-parallel applications. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016, Barcelona, Spain, March 12-16, 2016.* 44:1–44:2. https://doi.org/10.1145/2851141.2851193

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

[32] 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 *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015.* 619–630. https://doi.org/10.1145/2723372.2742786

[33] Jesús Escudero-Sahuquillo, Pedro Javier García, Francisco J. Quiles, German Maglione Mathey, and José Duato Marín. 2018. Feasible enhancements to congestion control in InfiniBand-based networks. *J. Parallel Distrib. Comput.* 112 (2018), 35–52. https://doi.org/10.1016/j.jpdc.2017.09.008

[34] Kapali P. Eswaran, Jim Gray, Raymond A. Lorie, and Irving L. Traiger. 1976. The Notions of Consistency and Predicate Locks in a Database System. *Commun. ACM* 19, 11 (1976), 624–633. https://doi.org/10.1145/360363.360369

[35] Zhisong Fu, Zhengwei Wu, Houyi Li, Yize Li, Min Wu, Xiaojie Chen, Xiaomeng Ye, Benquan Yu, and Xi Hu. 2017. GeaBase: A High-Performance Distributed Graph Database for Industry-Scale Applications. In *Fifth International Conference on Advanced Cloud and Big Data, CBD 2017, Shanghai, China, August 13-16, 2017.* 170–175. https://doi.org/10.1109/CBD.2017.37

[36] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012.* 17–30. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/gonzalez

[37] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.* 599–613. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/gonzalez

[38] Nusrat S. Islam, Md. Wasi-ur-Rahman, Jithin Jose, Raghunath Rajachandrasekar, Hao Wang, Hari Subramoni, Chet Murthy, and Dhabaleswar K. Panda. 2012. High performance RDMA-based design of HDFS over InfiniBand. In *SC Conference on High Performance Computing Networking, Storage and Analysis, SC '12, Salt Lake City, UT, USA - November 11 - 15, 2012.* 35. https://doi.org/10.1109/SC.2012.65

[39] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2014. Using RDMA efficiently for key-value services. In *ACM SIGCOMM 2014 Conference, SIGCOMM'14, Chicago, IL, USA, August 17-22, 2014.* 295–306. https://doi.org/10.1145/2619239.

2626299
[40] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. 185–201. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/kalia
[41] Supun Kamburugamuve, Karthik Ramasamy, Martin Swany, and Geoffrey C. Fox. 2017. Low Latency Stream Processing: Apache Heron with Infiniband & Intel Omni-Path. In *Proceedings of the 10th International Conference on Utility and Cloud Computing, UCC 2017, Austin, TX, USA, December 5-8, 2017*. 101–110. https://doi.org/10.1145/3147213.3147232
[42] H. T. Kung and John T. Robinson. 1979. On Optimistic Methods for Concurrency Control. In *Fifth International Conference on Very Large Data Bases, October 3-5, 1979, Rio de Janeiro, Brazil, Proceedings*. 351.
[43] Aapo Kyrola and Carlos Guestrin. 2014. GraphChi-DB: Simple Design for a Scalable Graph Database System - on Just a PC. *CoRR* abs/1403.0701 (2014). arXiv:1403.0701 http://arxiv.org/abs/1403.0701
[44] Matteo Lissandrini, Martin Brugnara, and Yannis Velegrakis. 2018. Beyond Macrobenchmarks: Microbenchmark-based Graph Database Evaluation. *PVLDB* 12, 4 (2018), 390–403. http://www.vldb.org/pvldb/vol12/p390-lissandrini.pdf
[45] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 135–146.
[46] Mugilan Mariappan, Joanna Che, and Keval Vora. 2021. DZiG: sparsity-aware incremental processing of streaming graphs. In *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*, Antonio Barbalace, Pramod Bhatotia, Lorenzo Alvisi, and Cristian Cadar (Eds.). ACM, 83–98. https://doi.org/10.1145/3447786.3456230
[47] Mugilan Mariappan and Keval Vora. 2019. GraphBolt: Dependency-Driven Synchronous Processing of Streaming Graphs. In *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*, George Candea, Robbert van Renesse, and Christof Fetzer (Eds.). ACM, 25:1–25:16. https://doi.org/10.1145/3302424.3303974
[48] Keith Marzullo and Susan S. Owicki. 1985. Maintaining the Time in a Distributed System. *Operating Systems Review* 19, 3 (1985), 44–54. https://doi.org/10.1145/850776.850780
[49] Christopher Mitchell, Yifeng Geng, and Jinyang Li. 2013. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *2013 USENIX Annual Technical Conference, San Jose, CA, USA, June 26-28, 2013*. 103–114. https://www.usenix.org/conference/atc13/technical-sessions/presentation/mitchell
[50] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. 2014. Scale-out NUMA. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, Salt Lake City, UT, USA, March 1-5, 2014*. 3–18. https://doi.org/10.1145/2541940.2541965
[51] Danica Porobic, Erietta Liarou, Pinar Tözün, and Anastasia Ailamaki. 2014. ATraPos: Adaptive transaction processing on hardware Islands. In *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*. 688–699. https://doi.org/10.1109/ICDE.2014.6816692
[52] David P. Reed. 1978. *Naming and synchronization in a decentralized computer system.* Ph.D. Dissertation. Massachusetts Institute of Technology, Cambridge, MA, USA. http://hdl.handle.net/1721.1/16279
[53] Semih Salihoglu and Jennifer Widom. 2013. Gps: A graph processing system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*. 22.
[54] Alon Shalita, Brian Karrer, Igor Kabiljo, Arun Sharma, Alessandro Presta, Aaron Adcock, Herald Kllapi, and Michael Stumm. 2016. Social Hash: An Assignment Framework for Optimizing Distributed Systems Operations on Social Networks. In *13th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2016, Santa Clara, CA, USA, March 16-18, 2016*, Katerina J. Argyraki and Rebecca Isaacs (Eds.). USENIX Association, 455–468. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/shalita
[55] Alex Shamis, Matthew Renzelmann, Stanko Novakovic, Georgios Chatzopoulos, Aleksandar Dragojevic, Dushyanth Narayanan, and Miguel Castro. 2019. Fast General Distributed Transactions with Opacity. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*. 433–448. https://doi.org/10.1145/3299869.3300069
[56] Wen Sun, Achille Fokoue, Kavitha Srinivas, Anastasios Kementsietsidis, Gang Hu, and Guo Tong Xie. 2015. SQLGraph: An Efficient Relational-Based Property Graph Store. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. 1887–1901. https://doi.org/10.1145/2723372.2723732
[57] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2012. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*. 1–12. https://doi.org/10.1145/2213836.2213838
[58] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. 2013. From think like a vertex to think like a graph. *Proceedings of the VLDB Endowment* 7, 3 (2013), 193–204.
[59] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*. 18–32. https://doi.org/10.1145/2517349.2522713
[60] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. 2016. PGQL: a property graph query language. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems, Redwood Shores, CA, USA, June 24 - 24, 2016*, Peter A. Boncz and Josep Lluís Larriba-Pey (Eds.). ACM, 7. https://doi.org/10.1145/2960414.2960421
[61] Keval Vora, Rajiv Gupta, and Guoqing Xu. 2017. KickStarter: Fast and Accurate Computations on Streaming Graphs via Trimmed Approximations. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017*, Yunji Chen, Olivier Temam, and John Carter (Eds.). ACM, 237–251. https://doi.org/10.1145/3037697.3037748
[62] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. 2015. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*. 87–104. https://doi.org/10.1145/2815400.2815419
[63] Ming Wu, Fan Yang, Jilong Xue, Wencong Xiao, Youshan Miao, Lan Wei, Haoxiang Lin, Yafei Dai, and Lidong Zhou. 2015. GraM: scaling graph computation to the trillions. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*. 408–421.
[64] Da Yan, Hongzhi Chen, James Cheng, Zhenkun Cai, and Bin Shao. 2018. Scalable De Novo Genome Assembly Using Pregel. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*. IEEE Computer Society, 1216–1219. https://doi.org/10.1109/ICDE.2018.00114
[65] Da Yan, Hongzhi Chen, James Cheng, M. Tamer Özsu, Qizhen Zhang, and John C. S. Lui. 2017. G-thinker: Big Graph Mining Made Easier and Faster. *CoRR* abs/1709.03110 (2017). arXiv:1709.03110 http://arxiv.org/abs/1709.03110
[66] Da Yan, James Cheng, Hongzhi Chen, Cheng Long, and Purushotham Bangalore. 2019. Lightweight Fault Tolerance in Pregel-Like Systems. In *Proceedings of the 48th International Conference on Parallel Processing, ICPP 2019, Kyoto, Japan, August 05-08, 2019*. ACM, 69:1–69:10. https://doi.org/10.1145/3337821.3337823
[67] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. 2014. Blogel: A block-centric framework for distributed computation on real-world graphs. *Proceedings of the VLDB Endowment* 7, 14 (2014), 1981–1992.
[68] Da Yan, Yuzhen Huang, Miao Liu, Hongzhi Chen, James Cheng, Huanhuan Wu, and Chengcui Zhang. 2018. GraphD: Distributed Vertex-Centric Graph Processing Beyond the Memory Limit. *IEEE Trans. Parallel Distributed Syst.* 29, 1 (2018), 99–114. https://doi.org/10.1109/TPDS.2017.2743708
[69] Erfan Zamanian, Carsten Binnig, Tim Kraska, and Tim Harris. 2016. The End of a Myth: Distributed Transactions Can Scale. *CoRR* abs/1607.00655 (2016). arXiv:1607.00655 http://arxiv.org/abs/1607.00655
[70] Jie Zhang, Xiaoyi Lu, and Dhabaleswar K. Panda. 2017. High-Performance Virtual Machine Migration Framework for MPI Applications on SR-IOV Enabled InfiniBand Clusters. In *2017 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2017, Orlando, FL, USA, May 29 - June 2, 2017*. 143–152. https://doi.org/10.1109/IPDPS.2017.43
[71] Qizhen Zhang, Akash Acharya, Hongzhi Chen, Simran Arora, Ang Chen, Vincent Liu, and Boon Thau Loo. 2019. Optimizing Declarative Graph Queries at Large Scale. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 1411–1428. https://doi.org/10.1145/3299869.3300064
[72] Qizhen Zhang, Hongzhi Chen, Da Yan, James Cheng, Boon Thau Loo, and Purushotham Bangalore. 2017. Architectural implications on the performance and cost of graph analytics systems. In *Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24-27, 2017*. ACM, 40–51. https://doi.org/10.1145/3127479.3128606
[73] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A Computation-Centric Distributed Graph Processing System. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, Kimberly Keeton and Timothy Roscoe (Eds.). USENIX Association, 301–316. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/zhu
[74] 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
[75] Tobias Ziegler, Sumukha Tumkur Vani, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. Designing Distributed Tree-based Index Structures for Fast RDMA-capable Networks. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*. 741–758. https://doi.org/10.1145/3299869.3300081