G-Tran: Making Distributed Graph Transactions Fast

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

Abstract

Graph transaction processing poses many unique challenges such as random data access due to the irregularity of graph structures, and high abort rate and low throughput due to the large read/write sets in graph transactions. To address these challenges, we present G-Tran — an RDMA-enabled distributed in-memory graph database. A graph-native data store is proposed to achieve good data locality and fast data access for transaction querying and updating. In addition, G-Tran adopts a fully decentralized architecture, which leverages RDMA to process distributed transactions, in order to avoid the performance bottlenecks caused by a traditional centralized coordinator. We also propose an MV-OCC protocol to address the issue of large read/write sets and to support both serializable and snapshot isolation. Extensive experiments show that G-Tran achieves competitive performance compared with other popular graph databases on benchmark workloads.

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 [1–4, 6, 9, 19, 25, 29, 38] have shortcomings in their design and functionality that lead to performance bottlenecks in processing complex graph transactions. For example, Titan [1], JanusGraph [2], OrientDB [6] and ArangoDB [4] follow a NoSQL-based schema as their underlying storage layout, but NoSQL focuses more on availability with relaxed consistency instead of ACID (i.e., Atomicity, Consistency, Isolation, Durability) for transaction processing. On the other hand, although RDBMSs can provide strong guarantee on ACID, the relational schema is not a good fit for graph transaction processing. Most graph queries/updates perform graph traversals and apply filters on vertices, edges and their properties. Such operations require expensive joins on relational tables, thus leading to serious performance bottlenecks especially when transactions involve multi-hop graph traversals [25, 29]. In view of this, Neo4j [3] and TigerGraph [19] adopt a *native graph* storage design, as to distinguish from the non-native storage designs (e.g., relational, columnar, key-value stores). Native graph databases store vertices/edges in adjacency-list structures (without indexing), which is more efficient for graph traversal. However, such storage layout results in poor data locality when updates are frequent; thus, it incurs extra overheads for data retrieval and impairs both latency and throughput (§3).

In addition to handling graph-specific operations, 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 for the following reasons. 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 and small-world phenomenon. Similarly, read-write transactions may also have large **write sets** (see Figure 6). 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 [1] motivate us to design a new distributed graph database system, called **G-Tran**, for high-performance and scalable transaction processing on **property graphs**. To the best of our knowledge, G-Tran is the first RDMA-based 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

[1]The unique challenges of graph transactions also distinguish the design objectives of graph databases (more details in §3) from those of graph processing systems such as Pregel [41], GraphX [31], PowerLyra [15], etc.

G-Tran as follows:

- 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 a data-parallel model.
- 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.
- 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 transaction processing.

We demonstrate the effectiveness of our system design and performance by comparing G-Tran with the state-of-the-art graph databases [2–4,19] using benchmark workloads [26,39]. The 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 to represent graph data because of the generality and 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. Many existing graph databases, such as Titan [1], JanusGraph [2], OrientDB [6] and Neo4J [3], adopt PG to model their graph data, while using *Gremlin* [5] as the query language. Currently, G-Tran also uses Gremlin (the latest 3.0 standard) as its query language.

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*) [28], *optimistic concurrency control* (*OCC*) [37], and *multiple version concurrency control* (*MVCC*) [47]. 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*. During the validation step, OCC runs conflict checking to determine if the current transaction should commit or 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 overhead on storage. In a distributed environment, transactions will need a global coordinator among workers to guide their potential validation and commit/abort (if any) based on the decisions collected. Usually, the coordinator is a unique node, which makes the entire system a centralized architecture. Consequently, the coordinator's performance and additional overhead on network round-trips for each transaction limit the throughput of a distributed database [40].

InfiniBand and RDMA. InfiniBand has become quite commonly in use in recent years and led to the development of many new distributed systems [27,32,35,59]. 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. RDMA's round-trip latency can be more than an order of magnitude lower than that of Ethernet and IPoIB.

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 structure can easily result in poor locality for data access after continuous updates. To address this issue, some existing systems store graph data in RDBMS, but this leads to the following issue.

(C2) Due to the power-law distribution on vertex degree and the small world phenomenon of most real-world graphs, the cost of processing traversal-based queries can be too high due to costly joins involving multiple-hop neighbors (e.g., $\geq$ 3 hop). This is the reason that native graph stores have been proposed recently, but this switches the problem back to (C1) again.

(C3) Although graph transaction workloads in real world are mostly read-heavy [9,22] (e.g., 99.8% of transactions handled by Facebook's Tao are read-only [9]), graph transactions typically have large read/write sets and long execution time,

which lead to high contention and result in low throughput.

(C4) Scalability is another critical problem for distributed graph transaction processing. The network bandwidth, contention likelihood, and CPU overheads are the main bottlenecks [58]. In addition, the traditional centralized architecture (i.e., assigning a master for global coordination) also limits the scalability.

We conduct experiments in §6 to verify the above issues. Here, we discuss the design choices to address the aforementioned 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 [8,16,23,33,44,49,53,58] have been proposed to improve the performance of distributed transaction processing by leveraging RDMA as it can remove the bottleneck of network, although these works mainly focus on RDBMS or key-value stores 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 we need to move from a pure shared-nothing or shared memory architecture to a hybrid one, i.e., shared-nothing local data management and remotely shared-memory via RDMA one-sided read/write and/or two-sided send/recv. These new features require us to have an overall system redesign from the storage layer to the transaction processing layer in order to tightly integrate the system components.

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 property 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 of G-Tran takes a hybrid approach, which splits the memory space of each machine into two parts. One part follows the shared-nothing architecture to store local graph data in partitions, and the other part follows the shared-memory architecture to compose an RDMA-based distributed memory space for remote data access. Such design is based on the facts that: (1) to implement efficient RDMA-based data structures (e.g., map, tree) are non-trivial, as the address space of each object should be recorded explicitly and this requires more memory footprint [61]; (2) not all data are necessary to be shared and managing different categories of data in different memory spaces compactly can improve data access efficiency for local search or scan. We provide more details about the memory layout and data placement in §4.2.

For (C3), 2PL would have a high overhead due to locking on large portions of data, i.e., large read/write sets. 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. Thus, we adopt a *multi-version optimistic concurrent control* (*MV-OCC*) 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 modification instead of ensuring consistency by acquiring locks. In addition, we further introduce two optimistic optimizations into the protocol to reduce the abort rate. We discuss the details of MV-OCC and the optimizations in §4.3.

Moreover, G-Tran adopts a decentralized architecture for transaction execution, validation and commit/abort (§4.4). Existing distributed databases [23, 25] adopt a centralized approach to process tasks such as transactional metadata management and timestamp ordering in a master node. The master is likely a bottleneck of the entire cluster and affects the overall performance [58]. Note that the processing load of validation and commit/abort can affect the latency of a graph transaction due to the large read/write sets, which accordingly increases the abort rate. In contrast, in our decentralized architecture, all servers share the load and overhead of any transaction by data-parallel processing according to the data locality of each sub-transaction. In addition, RDMA significantly reduces 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 1 shows the architecture of G-Tran. G-Tran supports multiple client connections to server nodes through a regular Ethernet network. Each client provides a user console and a lite communication lib for submitting transactions and receiving query results. A cluster monitor monitors the load of each server by a heartbeat mechanism and conducts the assignment of incoming transactions from clients for load balancing among servers. G-Tran server nodes are connected with each other via InfiniBand for RDMA communication.

As a distributed in-memory database, G-Tran stores all graph data, transaction metadata, and index structures in memory. Each server node is composed of two layers, i.e., the storage layer and the OLTP layer.

The storage layer, i.e., the data store, 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 over N server nodes. G-Tran constructs a consistent global address space over all the nodes in a cluster, so that the location of any object in the data store can be retrieved by its

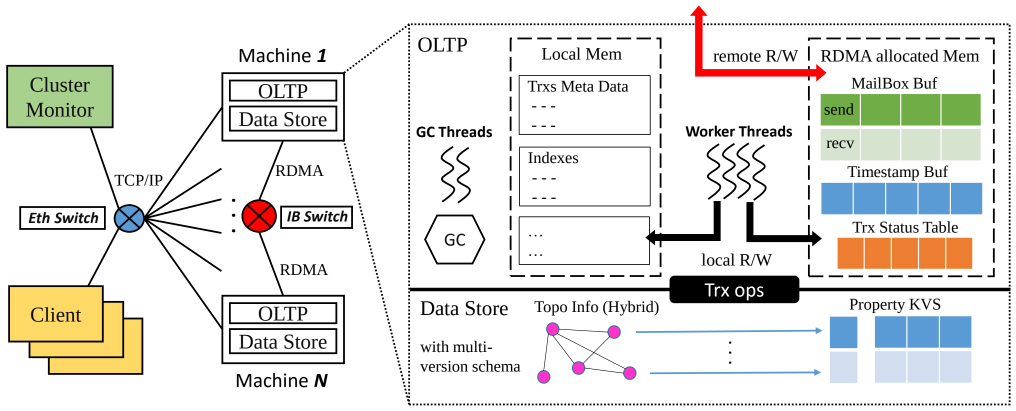


Figure 1: The architecture overview of G-Tran (best viewed in color)

ID. The data store has a multi-version schema to support a lock-free consistent view for concurrent transactions (§4.2).

The OLTP layer of each server node has a thread pool, which consists of worker threads to process incoming transactions. Worker threads interact with the data store to process data read/write. 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 1. 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 and fast remote data access and atomicity guarantee) for concurrent transaction processing.

G-Tran adopts a *data-parallel* model in the OLTP layer (§4.4), that is, a cross-server transaction is executed in parallel in the places where its data (i.e., the read/write sets) are located. The data-parallel model leads to better data locality and enables parallel execution of a single transaction. As a trade-off, parallel execution over multiple servers requires more message passing for consistency control, but we keep the network communication overhead low by using RDMA. We construct an RDMA-enabled mailbox (i.e., the green box in Figure 1) to process point-to-point message sending/receiving at thread-level, including one-sided RDMA read/write and two-sided RDMA send/recv. Besides the mailbox, in the RDMA allocated memory, G-Tran also maintains some system components (e.g., *timestamp bufs*, *transaction status table*) for global transactional metadata access, which is needed in our MV-OCC protocol (§4.3). However, the transactions' private metadata (e.g., begin time, commit time, read/write set), as well as other structures (e.g., indexes) that do not need to be shared with other servers, are stored in the local memory.

4.2 Data Store

As shown in Figure 2, 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).

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 2: *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

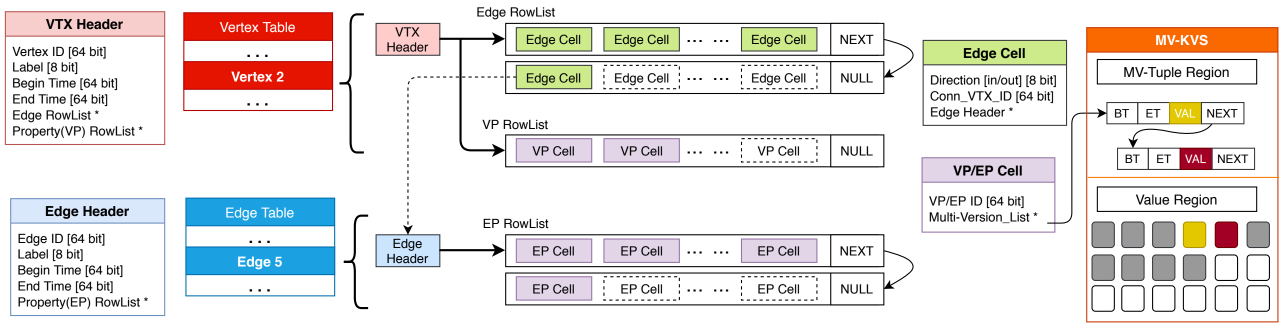


Figure 2: The data store of G-Tran (best viewed in color)

corresponding property. All MV-tuples (i.e., the different versions) of a property 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.

The Choice of Versioning. It is easy to understand the multi-versioning mechanism for property objects. For example, when a transaction updates (i.e., modify, insert, delete) the value of a vertex/edge property, G-Tran adds a new version to the MV-tuples of that property after its current version. But we do not apply multi-versioning to vertex/edge objects for the following reasons. If a transaction directly updates a vertex/edge object (i.e., the graph topology instead of the properties), this update will be directly executed on the corresponding Vertex and Edge Tables, rather than creating a new version for the vertex/edge. Because creating a new version for a vertex (or edge) requires a complete copy of the header, Edge RowList and VP RowList (or EP RowList), it would use much more space (but most of the data are actually duplicated) and processing time, leading to degraded transaction performance. This is also why we use begin/end time in the vertex/edge header to maintain the visible time period of a vertex/edge, such that a transaction can access/update a vertex/edge by checking/updating its begin/end time (§4.3).

Design Principle. One key design of the data store, as depicted in Figure 2, 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 data access. 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). With this data layout, graph traversals are executed as the combination of row-based scanning and object-based filtering (through the MV-KVS) under an RDMA one-sided read/write based message passing framework. The data layout also benefits the multi-threaded execution of concurrent transactions (§4.4).

4.3 The MV-OCC Protocol

To guarantee that a transaction T is serializable, we should hold the following two features in a multi-version storage. (1) *Read stability*: the visible version of a record for T should keep unchanged during the processing of T. Traditionally, this can be implemented by having a read lock on the record or by validating that there is no newly committed version of the record before T commits. (2) *Phantom avoidance*: the read set of T should keep unchanged during the processing of T. Usually, this can be implemented by having an index/row-level lock on the entire read set or by re-scanning the read set in a validation phase.

Ensuring serializability by locks and re-scanning has a high cost, especially in a distributed environment. To avoid locks and re-scanning, we propose our own MV-OCC design with specific optimizations to maintain isolation. Specifically, MV-OCC follows a general procedure of OCC while combining with the multi-version-based commit/abort rules. It separates execution from commit by first tracking the read/write sets of a transaction in the *execution phase*, validating the read set in the *validation phase*, and finally committing the write set in the *commit phase*, where the last two phases require atomicity. Figure 3 illustrates the workflow of a transaction in G-Tran. 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 in 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 updates 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 uses one-sided RDMA read to get the value.

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 such as their IDs, commit time and write sets. RCT is indexed by a B-Tree using the commit time as the key. RCT is used in the validation phase of each read-write transaction to avoid the high cost of read set re-scanning.

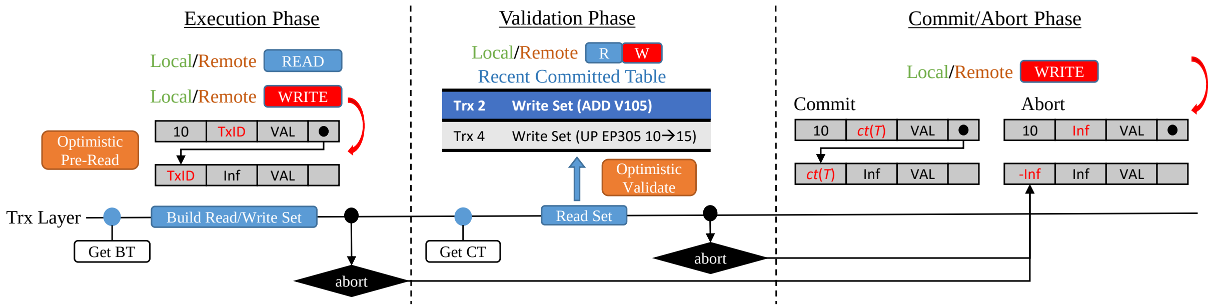


Figure 3: The MV-OCC protocol of G-Tran

Multi-Versioning Read/Write Rules. 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*) overlap 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., $TxID$) in 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 3. This indicates that this 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 in Figure 3. 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 three phases of the transaction workflow in Figure 3.

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 and its read/write set is constructed by accessing the Vertex/Edge Tables and the MV-KVS, 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 3, 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, 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:

- $TxID$'s status is *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.
- $TxID$'s status is *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.
- $TxID$'s status is *commit*: T directly reads V2 as $TxID$ has committed and the CT of $TxID$ is definitely earlier than the CT of T.
- $TxID$'s status is *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 obtain from RCT all the read-write transactions, *W-Trxs*, whose commit time falls in the period $[bt(T), ct(T)]$ and their write sets may change the read set of T. If there is no overlapping element between the read set of T and the write sets of *W-Trxs*, then T can commit successfully. Otherwise, T should abort. The above conflict checking is executed as set comparison in parallel on all server nodes (§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 all its dependent transactions due to optimistic pre-read have committed and all its dependent transactions due to optimistic validation have aborted. Note that if T is a read-only transaction, then its validation phase only needs to check those dependent transactions due to optimistic pre-read as 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 according to the multi-versioning read/write rules.

Design Principle. Following MV-OCC, there is no coarse-grained read/write locks or rescanning 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, for snapshot isolation, MV-OCC can skip optimistic pre-read in the execution phase and the entire validation phase, and performs the regular commit (or abort when write-write conflict happens).

4.4 Distributed Transaction Processing

We now describe how G-Tran leverages RDMA to process distributed transactions with the MV-OCC protocol. With the use of RDMA, the design goal of G-Tran's execution model is to effectively parallelize transaction processing across servers while removing the computation bottlenecks (e.g., centralized coordinator, stragglers, locks). Existing RDMA-enabled databases [16, 53, 58] usually apply RDMA atomic verbs, 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, as this incurs high CPU overheads and impairs transaction throughput. Instead of fetching data remotely for transaction processing, G-Tran processes transactions on the servers where the data is (i.e., *data-parallel*) and scales out transaction processing across the servers through message passing via one-sided RDMA write. The *data-parallel* model avoids extra CPU overheads incurred by remote locking and RDMA-based CAS & FAA.

G-Tran's decentralized architecture is shown as Figure 4, where all servers have the same layout and play the same roles in the cluster. Without loss of generality, we only discuss the processing of one single transaction T in one server. When a server receives T (assigned by the cluster monitor), this server becomes the unique *coordinator* of T and handles the transactional metadata/result management of T, i.e., to aggregate intermediate/final results of T if needed (e.g., for operators COUNT, MAX, etc.) and to update T's status in the local shard of *TST*.

In such a decentralized architecture, both TST and RCT are read/write-enabled, shared data structures via RDMA. In

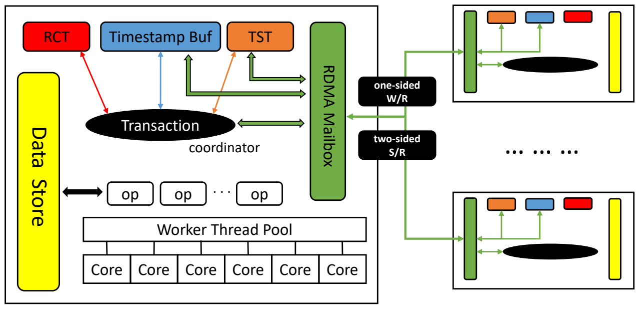


Figure 4: Distributed transaction processing in G-Tran

order to maintain the atomicity and consistency of TST/RCT read/write, we adopt two different strategies according to their specific workloads. Each update in TST is executed by a local atomic write on the coordinator server in order to avoid unsafe one-sided RDMA read from other server nodes. Applying atomic operation here is acceptable as a transaction status occupies only 2 bits. However, inserting an entire write set into RCT is a much bigger operation and implementing it on the RDMA allocated memory via RDMA-based CAS & FAA for read/write consistency incurs high CPU overhead and thread contention. Thus, we maintain RCT in the local memory of the servers, while using two-sided RDMA send/recv to query and update the RCT entries in order to ensure its read/write consistency. In addition, we maintain a *timestamp buf* in the RDMA allocated memory to synchronize the earliest BT among the active transactions on all servers through one-sided RDMA read, so as to obtain the global earliest BT, which is used by RCT to garbage-collect the expired entries (i.e., transactions and their write sets).

The entire processing of a transaction T is split into shards cross all servers and executed in parallel. First, the coordinator of T sends initial commands to all servers via one-sided RDMA write to trigger the start of T's processing. But not every server can be activated for T, as it depends on the actual data locality of the read set of T. If a server indeed has one shard of T, then it follows the MV-OCC protocol to process T. 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 every shard of TST on the servers via one-sided RDMA write. Thus, at the beginning of each operation in T's processing, the first step we have 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 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 the potential conflicting transactions through RCT, and then does conflict checking on their read/write sets as described in §4.3. Finally, if T can commit successfully, besides following the procedures of MV-OCC in §4.3, the final result will be aggregated from each server to T's coordinator and then sent to the client.

The data-parallel model leverages RDMA to enable efficient distributed transaction processing on the local data of each server to achieve better data locality. We also take into consideration the side-effects caused by frequent thread switching and from the NUMA architecture [24, 45]. As shown in Figure 4, each thread in our *worker thread pool* is bound with one CPU core and accordingly we divide these threads into several logical thread regions, where each region processes only one category of operators distinguished by their functionalities and data access patterns. For example, the operators COUNT, MAX and MIN belong to the *aggregation* category, and the operators IN, OUT and BOTH belong to the *traversal* category. This design can achieve better cache locality and memory locality on CPUs in a cross-NUMA node for data processing, which is motivated by Gemini [60] and Grasper [12]. 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. To support RDMA communication, we developed a mailbox to achieve efficient RDMA one-sided read/write and two-sided send/recv based on RDMA *ibverbs* library. In addition, we also provide a general TCP-based version of the system, i.e., G-Tran without RDMA, which uses ZeroMQ TCP sockets to achieve point-to-point communication and MPI to coordinate the inter-process communication over Ethernet. Here we briefly discuss some of the system implementation details, while all code and details will be released on GitHub later.

Timestamp Ordering. Unified timestamp (together with MV-OCC) allows concurrent transactions to read a consistent snapshot of the database, while the timestamp order should be matched with the real time order. We follow the solution proposed by FaRMv2 [49] for global time synchronization using Marzullo's algorithm [43], 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 then execute GC 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 Appendix B.

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. When read-write transactions commit, the indexes will also be updated in order to maintain consistency.

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 [3] and TigerGraph Developer Edition [19]. 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 used the same number of computing threads in each machine 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[2] data generator, and two real-world property graphs crawled from DBpedia[3] (including two parts, citation data and citation links) and Amazon Product[4] (including product reviews and metadata). The small/large datasets were used for the system evaluation on the single-machine/distributed environment respectively. Table 1 lists the statistics of each dataset.

Query Benchmarks. Lissandrini, Brugnara and Velegrakis [39] proposed a query benchmark (denoted as **LBV**) for graph database evaluation, which includes five categories of queries: *Creation*, *Read*, *Update*, *Deletion* and *Traversal*. *Read* and

[2]http://ldbcouncil.org/developer/snb

[3]https://wiki.dbpedia.org/downloads-2016-10

[4]http://jmcauley.ucsd.edu/data/amazon/

Table 1: Property graph datasets

Dataset	$\|V\|$	$\|E\|$	$\|VP\|$	$\|EP\|$
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

Traversal belong to *READ* transactions, while *Creation*, *Update* and *Deletion* belong to *WRITE* transactions. We selected, with equal probability, most of the queries in the benchmark (except few that are not for transactions, e.g., counting vertices) for throughput evaluation. We also used 8 queries, *Interactive Complex* IC1-IC4 and *Interactive Short* IS1-IS4, in the **LDBC SNB** benchmark [26], as these queries are more complex and we used them for single-query latency evaluation. We list the templates of the queries in both the LBV and LDBC benchmarks in Appendix C.

6.1 Evaluation of System Designs

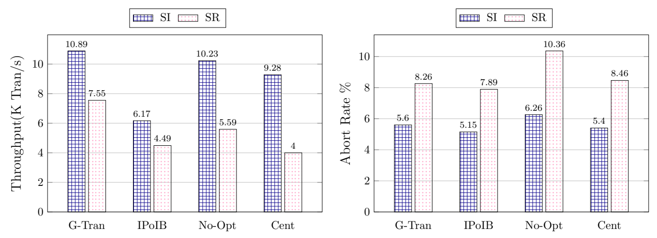
We first evaluate the effectiveness of the various system designs, including the data store, the decentralized architecture, the two optimizations in the MV-OCC protocol (i.e., optimistic pre-read and optimistic validation), and the speed-up due to RDMA-related designs.

6.1.1 Evaluation of Individual System Designs

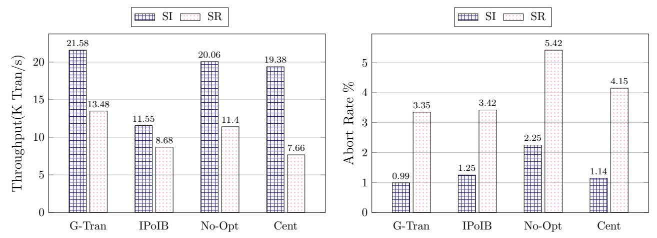
To examine the effect of each individual design on the system performance, we created three variants of G-Tran: G-Tran without RDMA but using IPoIB (denoted as **IPoIB**), G-Tran without the two optimizations in MV-OCC (denoted as **No-Opt**), and a centralized version (denoted as **Cent**) with a global master (i.e., transaction coordinator). We tested them on two large graphs, LDBC-L and Amazon, using 8 machines with 20 computing threads per machine. 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 5 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.

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 5(a), on the LDBC-L dataset, No-Opt's throughput (4.49K 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, which leads to higher contention and more transactions are aborted when conflicts increase. The result thus shows that the optimizations in the MV-OCC protocol can effectively improve the processing of concurrent read/write graph transactions that have large read/write-sets. We report the distribution of the sizes of the read/write-sets of the transactions in Figure 6, which shows that although more transactions have relatively smaller read/write-sets, there are also a large number of transactions that have large read/write-sets. This also explains the relatively high abort rate of G-Tran at SR compared with that at SI.



(a) LDBC-L



(b) Amazon

Figure 5: The effects of individual system design

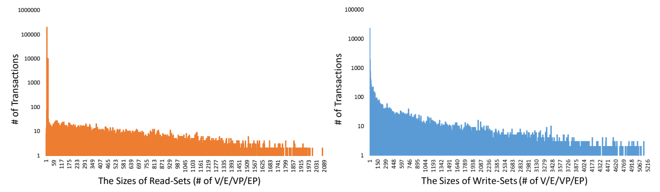


Figure 6: The distribution of the sizes of the read/write-sets of the LBV graph transactions on Amazon

To demonstrate the advantage of the decentralized architecture in the RDMA network environment for distributed transaction processing, we further compare G-Tran with its Cent variant. At SR isolation, Cent has the lowest throughput for both datasets (i.e, 4K tps and 7.66K tps). The main reason for Cent's poor performance is that, in a centralized architecture, the master plays the role of a global coordinator and needs to handle the coordinating tasks for all concurrent transactions in all servers, as well as the specific maintenance for the centralized RCT and TST, and the timestamp assignment. 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 variant compared with G-Tran, as the increased average latency per transaction leads to higher contention and in turn increases the overall abort rate.

We also note that all the variants at SI obtain higher throughput and lower abort rate than at SR, because concurrent transactions at SR need to ensure logically sequential execution, in which case the contention becomes intense.

Table 2: 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	96,717	1,249	1.1	25.4	6.8	1.6
J.G.	198,652	14,452	1.3E6	125,807	1.2	20.6	2.7	0.9
ArangoDB	138,810	1,420	91,584	3,149	1.1	58.9	33.6	0.8
T.G.(install	45,460	41,044	44,275	45,126	38,367	39,029	36,782	35,973
+ run)	+63.5	+19.1	+370	+21.3	+8.2	+15.2	+8.9	+6.7

Table 3: The latency (in *msec*) of the *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	40,518	59,968	92,770	132,766
+ run)	+8.82	+389	+3,788	+62,053

6.1.2 Evaluation of Data Store

Next, we evaluate how the design of the data store affects 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 of G-Tran, which allows us to exclude the influence of G-Tran's RDMA-aware components and decentralized architecture in the analysis of the data store's effectiveness. In addition, before we started to run the workloads in this experiment, we first warmed up G-Tran (and the systems we compared with) by running the mixed LBV workload in §6.1.1 for an extended period of time, so as to simulate a real scenario in which the locality of the stored graph data (e.g., connected vertices) might be broken after continuous updates by the transactions. The broken data locality of graph data after frequent updates is well-known to lead to more random access on the entities of the graph, which can thus be used to indicate the effectiveness of the data layout.

We used the 8 complex benchmark queries, IC1-IC4 and IS1-IS4, in LDBC [26] in order to examine the efficiency of supporting more complex data access patterns. As these benchmark queries can only function on the LDBC synthetic dataset, we also used a typical multi-hop query workload that consists of four graph-traversal-based query templates with the format: $g.V().has([primary\_key]).(both())^k$. The *both()* operator scans and returns both the in-neighbors and out-neighbors of a source vertex. Here, *both()* is repeated $k = 1, 2, 3, 4$ times to represent a k-hop traversal from a starting vertex, which is filtered by the *has()* operator with a primary key (e.g., name) on the vertex property.

We compared G-Tran with JanusGraph, ArangoDB, Neo4J and TigerGraph, where all the systems ran on a single machine using 20 threads. Table 2 and Table 3 report the latency of processing the LDBC queries and the *k*-hop traversal queries, respectively. JanusGraph failed to load DBpedia in 24 hours. TigerGraph also failed as it requires a fixed schema but many real-world graphs such as DBPedia have no fixed schema.

G-Tran achieves the shortest latency for processing IC1-IC4 and IS1-IS4. 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 3 shows that the latency of all the systems increases exponentially as *k* increases. This is because the size of the read-set grows exponentially for each hop of traversal on the graph. But after traversing more than 2 hops of neighbors, G-Tran starts to show orders of magnitude improvement over the other systems.

We explain the results by analyzing the designs of each system related to data access as follows. Although both Neo4J and TigerGraph have a native data store tailored for graph data, Neo4J stores all edges of the entire graph in a global sequential table and TigerGraph stores edges separately based on their types under a fixed schema. Consequently, Neo4J's data layout suffers from jump addressing in the physical storage space during a graph traversal, as edge insertions are written at the end of the sequential table like write-ahead logging (WAL). TigerGraph is not open source, but as it requires users to indicate the type (e.g., friendship) of an edge during a traversal, we conjecture that it stores edges of the same type together. This layout requires an extra aggregation to merge those separate edge sets when traversals involve multiple or all edge types, which is probably why TigerGraph has poor performance for the *k*-hop traversal queries. Both JanusGraph and ArangoDB follow a typical NoSQL-based format to store relations (i.e., edges) as individual cells or collections. Specifically, JanusGraph stores edge cells together with the property cells in the same row for each vertex, which results in extra overhead in locating edge objects from the property cells. ArangoDB stores edges as collections of documents. Thus, the traversal of each hop needs to locate the connected edges of each vertex by searching the edge collections.

In comparison, G-Tran divides arbitrary-length adjacency-lists into fixed-size rows, which supports more efficient edge insertion and deletion by inserting new rows or merging under-filled rows. Note that a new row is inserted only after filling up the blanks in an existing row and we use lazy deletion and periodically merge sparse rows. It also stores the property values separately in a key-value store. As a result, traversals can access the edges of a vertex sequentially and the properties of a vertex can also be efficiently accessed, even after frequent updates on the graph.

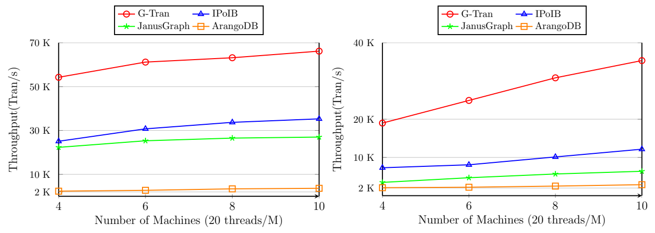
6.2 Throughput Analysis

Throughput is a major performance metric of transaction processing. In this set of experiments, we compared G-Tran with existing systems on their transaction throughput in both distributed and single-machine settings. We used three workloads: (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

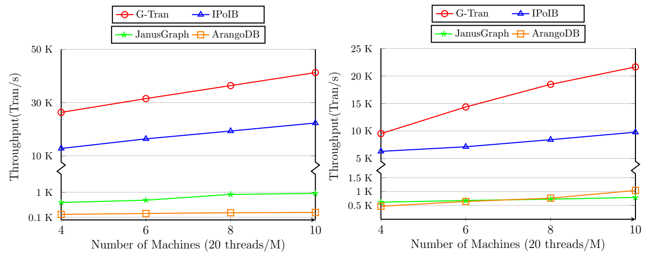
We first compared G-Tran and its IPoIB variant (G-Tran-IPoIB) with JanusGraph (using HBase as the storage backend) and ArangoDB. For Neo4J and TigerGraph, we only have their Developer Edition which does not support distributed processing. We set isolation at the SI level as both ArangoDB and distributed JanusGraph do not support SR. Figures 7-9 report the throughput of the systems on the three workloads, respectively, running on 4 to 10 machines with 20 threads/machine. Similar throughput performance is also observed using 8 machines but increasing the number of threads from 4 to 20, which is reported in Appendix A.

G-Tran achieves significantly higher throughput than the other systems for all the three workloads. Although the performance of G-Tran-IPoIB is degraded as RDMA features are disabled, it still outperforms both JanusGraph and ArangoDB in all cases. This further confirms 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 as 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 for processing the heavier workloads (i.e., RI, WI). In contrast, the throughput of both JanusGraph and ArangoDB are relatively low and scales poorly with the increase in the number of machines. In addition to their inefficient data storage as we discussed in §6.1.2, this is also caused by the partitioning strategy and data representation of JanusGraph and ArangoDB. Specifically, JanusGraph uses random partitioning to assign vertices to machines by default, which breaks the locality of the graph and consequently leads to extra overhead on communication. ArangoDB organizes vertices and edges into separate collections and connecting vertices to their edges requires expensive



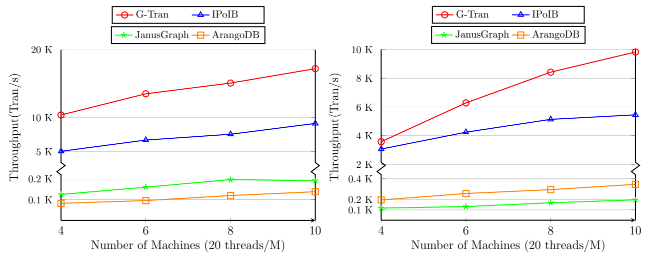
(a) Amazon (b) LDBC-L

Figure 7: Scale-out throughput on Read-Only workload



(a) Amazon (b) LDBC-L

Figure 8: Scale-out throughput on Read-Intensive workload



(a) Amazon (b) LDBC-L

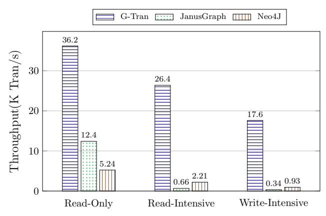
Figure 9: Scale-out throughput on Write-Intensive workload

joins based on the vertex IDs. In contrast, G-Tran's data store (as shown in Figure 2) keeps direct connection between vertices and edges and also stores them together in each machine.

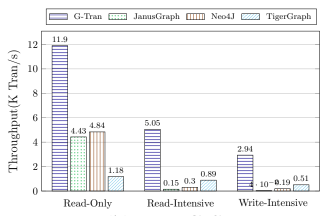
6.2.2 Single-Machine Processing

We compared G-Tran with Neo4J, TigerGraph, JanusGraph and ArangoDB for single-machine transaction throughput at both SI and SR. We used BerkeleyDB as the backend of JanusGraph for its evaluation at SR, as only BerkeleyDB supports SR as a single-machine system. Neo4J and TigerGraph do not support SI, while ArangoDB does not support SR. TigerGraph and JanusGraph failed to load DBPedia as explained in §6.1.2. We used 20 threads for all the systems.

As reported in Figure 10 and Figure 11, G-Tran achieves significantly higher throughput than the other systems for all workloads at both SR and SI. We note that here in the single-machine setting, the performance advantages of G-Tran do not come from RDMA and its decentralized architecture, but mainly because of its data store design and MV-OCC protocol.

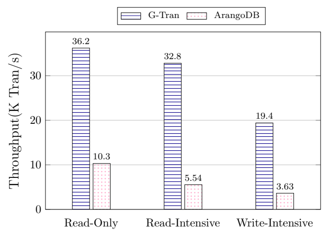


(a) DBPedia

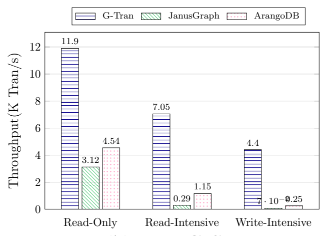


(b) LDBC-S

Figure 10: Single-machine throughput at SR



(a) DBPedia



(b) LDBC-S

Figure 11: Single-machine throughput at SI

At SR isolation, JanusGraph's performance is much worse than that of G-Tran, TigerGraph and Neo4J for the RI and WI workloads, although it achieves reasonable throughput for the RO workload. This is because G-Tran, TigerGraph and Neo4J have more efficient native graph stores with more tailored designs to handle transactions, while JanusGraph is built on a general NoSQL-based store. Thus, TigerGraph and Neo4J are better at processing more write-intensive concurrent transactions than JanusGraph, but they have higher overheads for reads. In comparison, G-Tran achieves high throughput for both write and read intensive workloads at both SR and SI, thanks to its multi-version based storage that enables lock-free snapshot reads and its data-parallel execution engine that enables parallel processing inside each transaction. In addition, compared with TigerGraph and Neo4J, G-Tran further has its own optimizations in the MV-OCC protocol to reduce the abort rate of concurrent read-write transactions. Neo4J also requires explicit locks issued by a user query to achieve SR support, and TigerGraph has a costly "install" stage to pre-translate and optimize queries before their real execution. In contrast, G-Tran requires neither explicit lock nor "install" for transaction processing with SR guarantee.

At the SI level, ArangoDB achieves better performance than JanusGraph due to its MVCC-based storage, leading to lower overhead for transactional guarantee. However, due to its inefficient graph data storage, ArangoDB's throughput is still significantly lower than that of G-Tran.

7 Related Work

Graph Databases. Neo4J [3] is one of the most popular graph databases. It supports ACID with *read committed* isolation, but requires users to use a query language to issue explicit locks on entities to enable higher isolation levels (e.g., SR). Titan [1] and JanusGraph [2] adopt a NoSQL-based storage for graph transaction processing. However, they have limited ACID support, only when using BerkeleyDB as the backend can they provide full SR isolation in a single-machine setting. Both ArangoDB [4] and OrientDB [6] use a multi-model document-based store for graph data, while ArangoDB supports *snapshot* isolation and OrientDB supports *read committed* and *repeatable reads* isolation. Graphflow [36] is a prototype graph database focusing on subgraph query workloads, and two worst-case optimal join algorithms are proposed to achieve efficient one-time and continuous subgraph queries. TigerGraph [19] is a popular graph database that supports ACID and aims at massively parallel processing for both graph transaction processing and graph analytics. Compared with these existing systems, G-Tran has distinguished designs in its graph-native data store, RDMA-featured decentralized architecture and MV-OCC, which contribute to its high performance as reported in §6.

Graph Processing Systems. Many systems have been proposed for offline graph computation [7,15,30,31,48,54,57,60] following Pregel [41]. 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 [13, 14, 51, 55, 56], or streaming graph processing [11,20,42]. These systems are not designed for and hence cannot handle online graph transaction workloads.

Distributed Transaction Processing. Many systems have been proposed in recent years for distributed transaction processing, including Google's Spanner [17], Granola [18], FaSST [34] and others [21,46,50,52]. Some of them leverage new hardwares such as RDMA, HTM and NVM to achieve high performance (e.g., FaRM [22,23] and DrTM [16,53]). 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 [10] and RDMA to ensure the strong consistency and atomicity. However, these systems are not specially designed for graph data, which has its own unique challenges as we discussed in §1 and §3.

8 Conclusions

We presented G-Tran, a high-performance RDMA-based graph database with both serializability and snapshot isolation support. To tackle the unique challenges of graph transaction processing, we used RDMA one-sided/two-sided verbs to reduce system overheads from network and CPUs. G-Tran also proposed its own data layout, transaction protocol and decentralized architecture to achieve an overall good performance in terms of both latency and throughput. For future work, we plan to enhance G-Tran on durability and availability by addressing fault tolerance and data replication.

References

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

[2] JanusGraph, 2017. http://janusgraph.org/.

[3] Neo4J, 2018. https://neo4j.com/.

[4] ArangoDB, 2019. https://www.arangodb.com/.

[5] Gremlin, 2019. https://tinkerpop.apache.org/gremlin.html.

[6] OrientDB, 2019. https://orientdb.com/.

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

[8] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. The end of slow networks: It's time for a redesign. *PVLDB*, 9(7):528–539, 2016.

[9] 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. TAO: facebook's distributed data store for the social graph. In *2013 USENIX Annual Technical Conference, San Jose, CA, USA, June 26-28, 2013*, pages 49–60, 2013.

[10] Trevor Brown and Hillel Avni. Phytm: Persistent hybrid transactional memory. *PVLDB*, 10(4):409–420, 2016.

[11] Cheng Chen, Hejun Wu, Dyce Jing Zhao, Da Yan, and James Cheng. Sgraph: A distributed streaming system for processing big graphs. In Yu Wang, Ge Yu, Yanyong Zhang, Zhu Han, and Guoren Wang, editors, *Big Data Computing and Communications - Second International Conference, BigCom 2016, Shenyang, China, July 29-31, 2016. Proceedings*, volume 9784 of *Lecture Notes in Computer Science*, pages 285–294. Springer, 2016.

[12] Hongzhi Chen, Changji Li, Juncheng Fang, Chenghuan Huang, James Cheng, Jian Zhang, Yifan Hou, and Xiao Yan. 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*, pages 87–100. ACM, 2019.

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

[14] Hongzhi Chen, Xiaoxi Wang, Chenghuan Huang, Juncheng Fang, Yifan Hou, Changji Li, and James Cheng. 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.*, pages 1881–1884, 2019.

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

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

[17] 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. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.*, 31(3):8:1–8:22, 2013.

[18] James A. Cowling and Barbara Liskov. Granola: Low-overhead distributed transaction coordination. In *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*, pages 223–235, 2012.

[19] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor Lee. Tigergraph: A native MPP graph database. *CoRR*, abs/1901.08248, 2019.

[20] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. Low-latency graph streaming using compressed purely-functional trees. In Kathryn S. McKinley and Kathleen Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 918–934. ACM, 2019.

[21] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Åke Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 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*, pages 1243–1254, 2013.

[22] Aleksandar Dragojevic, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 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*, pages 401–414, 2014.

[23] Aleksandar Dragojevic, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 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*, pages 54–70, 2015.

[24] Andi Drebes, Antoniu Pop, Karine Heydemann, Nathalie Drach, and Albert Cohen. 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*, pages 44:1–44:2, 2016.

[25] Ayush Dubey, Greg D. Hill, Robert Escriva, and Emin Gün Sirer. Weaver: A high-performance, transactional graph database based on refinable timestamps. *PVLDB*, 9(11):852–863, 2016.

[26] Orri Erling, Alex Averbuch, Josep-Lluís Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat-Pérez, Minh-Duc Pham, and Peter A. Boncz. 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*, pages 619–630, 2015.

[27] Jesús Escudero-Sahuquillo, Pedro Javier García, Francisco J. Quiles, German Maglione Mathey, and José Duato Marín. Feasible enhancements to congestion control in infiniband-based networks. *J. Parallel Distrib. Comput.*, 112:35–52, 2018.

[28] Kapali P. Eswaran, Jim Gray, Raymond A. Lorie, and Irving L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11):624–633, 1976.

[29] Zhisong Fu, Zhengwei Wu, Houyi Li, Yize Li, Min Wu, Xiaojie Chen, Xiaomeng Ye, Benquan Yu, and Xi Hu. 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*, pages 170–175, 2017.

[30] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 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*, pages 17–30, 2012.

[31] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.*, pages 599–613, 2014.

[32] Nusrat S. Islam, Md. Wasi-ur-Rahman, Jithin Jose, Raghunath Rajachandrasekar, Hao Wang, Hari Subramoni, Chet Murthy, and Dhabaleswar K. Panda. 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*, page 35, 2012.

[33] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using RDMA efficiently for key-value services. In *ACM SIGCOMM 2014 Conference, SIGCOMM'14, Chicago, IL, USA, August 17-22, 2014*, pages 295–306, 2014.

[34] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 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.*, pages 185–201, 2016.

[35] Supun Kamburugamuve, Karthik Ramasamy, Martin Swany, and Geoffrey C. Fox. Low latency stream processing: Apache heron with infiniband & intel omnipath. In *Proceedings of the 10th International Conference on Utility and Cloud Computing, UCC 2017, Austin, TX, USA, December 5-8, 2017*, pages 101–110, 2017.

[36] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedbhi, Jeremy Chen, and Semih Salihoglu. Graphflow: An active graph database. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 1695–1698, 2017.

[37] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. In *Fifth International Conference on Very Large Data Bases, October 3-5, 1979, Rio de Janeiro, Brazil, Proceedings.*, page 351, 1979.

[38] Aapo Kyrola and Carlos Guestrin. Graphchi-db: Simple design for a scalable graph database system - on just a PC. *CoRR*, abs/1403.0701, 2014.

[39] Matteo Lissandrini, Martin Brugnara, and Yannis Velegrakis. Beyond macrobenchmarks: Microbenchmark-based graph database evaluation. *PVLDB*, 12(4):390–403, 2018.

[40] Yi Lu, Xiangyao Yu, and Samuel Madden. STAR: scaling transactions through asymmetrical replication. *CoRR*, abs/1811.02059, 2018.

[41] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146, 2010.

[42] Mugilan Mariappan and Keval Vora. Graphbolt: Dependency-driven synchronous processing of streaming graphs. In George Candea, Robbert van Renesse, and Christof Fetzer, editors, *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*, pages 25:1–25:16. ACM, 2019.

[43] Keith Marzullo and Susan S. Owicki. Maintaining the time in a distributed system. *Operating Systems Review*, 19(3):44–54, 1985.

[44] Christopher Mitchell, Yifeng Geng, and Jinyang Li. 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*, pages 103–114, 2013.

[45] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. Scale-out NUMA. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, Salt Lake City, UT, USA, March 1-5, 2014*, pages 3–18, 2014.

[46] Danica Porobic, Erietta Liarou, Pinar Tözün, and Anastasia Ailamaki. 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*, pages 688–699, 2014.

[47] David P. Reed. *Naming and synchronization in a decentralized computer system*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1978.

[48] Semih Salihoglu and Jennifer Widom. Gps: A graph processing system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, page 22, 2013.

[49] Alex Shamis, Matthew Renzelmann, Stanko Novakovic, Georgios Chatzopoulos, Aleksandar Dragojevic, Dushyanth Narayanan, and Miguel Castro. 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.*, pages 433–448, 2019.

[50] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 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*, pages 1–12, 2012.

[51] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. From think like a vertex to think like a graph. *Proceedings of the VLDB Endowment*, 7(3):193–204, 2013.

[52] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 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*, pages 18–32, 2013.

[53] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. 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*, pages 87–104, 2015.

[54] Ming Wu, Fan Yang, Jilong Xue, Wencong Xiao, Youshan Miao, Lan Wei, Haoxiang Lin, Yafei Dai, and Lidong Zhou. Gram: scaling graph computation to the trillions. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 408–421, 2015.

[55] Da Yan, Hongzhi Chen, James Cheng, M. Tamer Özsu, Qizhen Zhang, and John C. S. Lui. G-thinker: Big graph mining made easier and faster. *CoRR*, abs/1709.03110, 2017.

[56] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. Blogel: A block-centric framework for distributed computation on real-world graphs. *Proceedings of the VLDB Endowment*, 7(14):1981–1992, 2014.

[57] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. Effective techniques for message reduction and load balancing in distributed graph computation. In *Proceedings of the 24th International Conference on World Wide Web*, pages 1307–1317, 2015.

[58] Erfan Zamanian, Carsten Binnig, Tim Kraska, and Tim Harris. The end of a myth: Distributed transactions can scale. *CoRR*, abs/1607.00655, 2016.

[59] Jie Zhang, Xiaoyi Lu, and Dhabaleswar K. Panda. 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*, pages 143–152, 2017.

[60] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A computation-centric distributed graph processing system. In Kimberly Keeton and Timothy Roscoe, editors, *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 301–316. USENIX Association, 2016.

[61] Tobias Ziegler, Sumukha Tumkur Vani, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 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.*, pages 741–758, 2019.

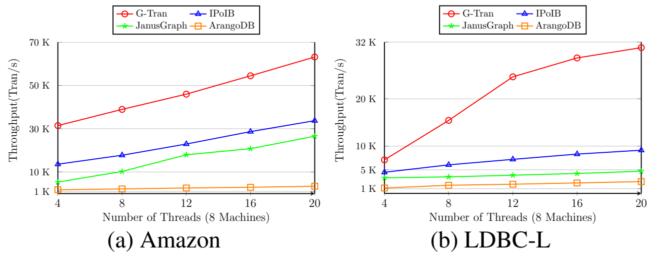


Figure 12: Scale-up throughput on Read-Only workload

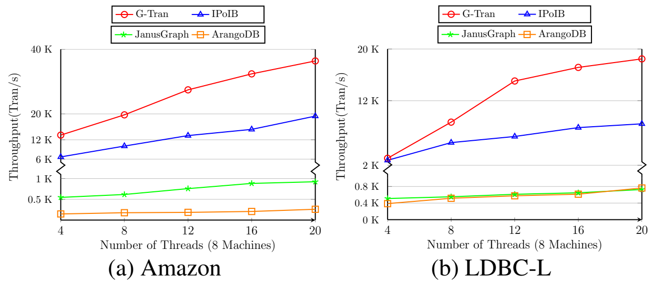


Figure 13: Scale-up throughput on Read-Intensive workload

A Scale-Up Throughput

In §6.2.1 we have reported the scale-out throughput performance of G-Tran, comparing with JanusGraph and ArangoDB. Here, we also report the scale-up throughput performance of the systems on the same RO, RI and WI workloads. Figure 12, Figure 13 and Figure 14 report the results, where all the systems used 8 machines and we varied the number of threads in each machine from 4 to 20.

G-Tran achieves significantly higher throughput than the other systems, including its own IPoIB variant. G-Tran also achieves much better vertical scalability than JanusGraph and ArangoDB, which we explain as follows. With more threads on each machine, G-Tran's engine assigns higher parallelism to the processing of the transactions that are currently being processed. As the result, these transactions can commit earlier than before. Consequently, there is less contention in accessing data, which reduces the overhead of concurrent transaction processing. In contrast, both JanusGraph and ArangoDB follow the one-thread-one-transaction mechanism. When more threads are available, more transactions are accepted into the system and being processed simultaneously. While this allows the system to process more transactions, it also increases the contention among transactions and incurs more overhead for concurrent transactions. Thus, the throughput of JanusGraph and ArangoDB does not increase much when we use more threads. This phenomenon are more obvious on the Read-Intensive and Write-Intensive workloads.

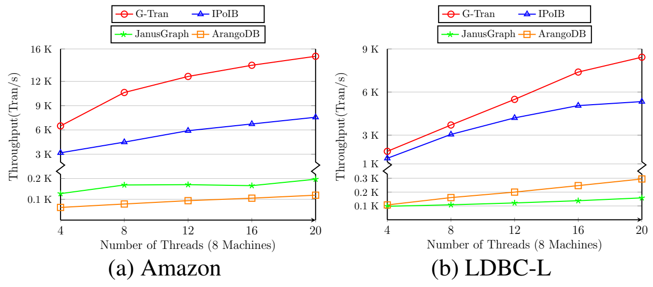


Figure 14: Scale-up throughput on Write-Intensive workload

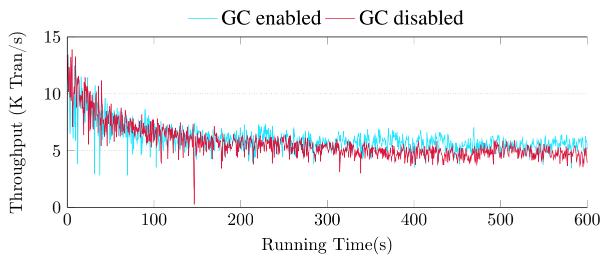


Figure 15: Throughput with GC enabled/disabled

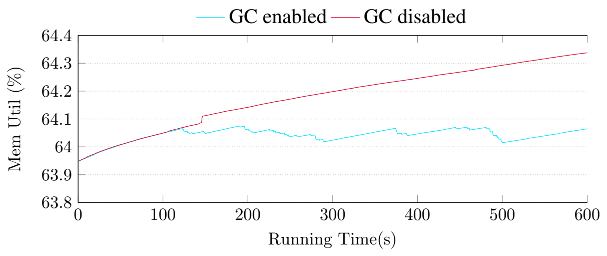


Figure 16: Memory consumption with GC enabled/disabled

B Performance of Garbage Collection

We also evaluated the effectiveness of G-Tran's Garbage Collection (GC) mechanism by comparing the performance difference between enabling and disabling GC during regular transaction processing. We conducted the experiment using 8 machines with 20 computing threads per machine on the LDBC-L dataset and executed the same workload at SR isolation as we did in §6.1.1.

Figure 15 and Figure 16 report 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 16, which are as a result of 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 the two curves in Figure 15, 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.

As the GC mechanism categorizes and packages different GC objects into different types of tasks, the result demonstrates the effectiveness of G-Tran's GC mechanism for recycling various types of obsolete objects in a timely fashion (as otherwise the memory footprint would increase linearly over time). The result also shows that G-Tran's GC mechanism is also efficient as the throughput is almost not affected during the whole process.

C Benchmark Query Templates

We list the benchmark queries used in our experimental evaluation in §6. Table 4 lists the query templates for the **LBV** benchmark [39] and and Table 5 lists the query templates for the **LDBC SNB** benchmark [26]. The values of the vertex/edge id and properties are randomly sampled from the respective datasets on which the queries are evaluated.

Table 4: Query templates in the **LBV** benchmark

Q2	g.addVertex(p[]) // create a new vertex with properties p
Q3	g.addEdge(v1, v2, l) // add an edge with label *l* from *v*1 to *v*2
Q4	g.addEdge(v1, v2, l, p[])
Q5	v.setProperty(Name, Value)
Q6	e.setProperty(Name, Value)
Q7	g.addVertex(...); g.addEdge(...) // add a new vertex, and then edges to the vertex
Q11	g.V.has(Name, Value) // Vertices with property *Name* = *Value*
Q12	g.E.has(Name, Value)
Q13	g.E.has("label", l)
Q14	g.V(id) // The vertex with identifier *id*
Q15	g.E(id)
Q16	v.setProperty(Name, Value)
Q17	e.setProperty(Name, Value)
Q18	g.removeVertex(id)
Q19	g.removeEdge(id)
Q20	v.removeProperty(Name)
Q21	e.removeProperty(Name)
Q22	v.in() // Vertices adjacent to *v* via incoming edges
Q23	v.out()
Q24	v.both("l") // Vertices adjacent to *v* via edges labeled *l*
Q25	v.inE.label.dedup() // Labels of incoming edges of *v* (no dupl.)
Q26	v.outE.label.dedup()
Q27	v.bothE.label.dedup()

Table 5: Query templates in the **LDBC SNB** benchmark

IS1	g.V().has("ori_id", "ID").union(properties("firstName", "lastName", "birthday", "locationIP", "browserUsed"), out("isLocatedIn").properties("name"))

IS2	g.V().has("ori_id", "ID").in("hasCreator")
	.order("creationDate", decr).limit(10).as("msg").union(
	hasLabel("post").out("hasCreator").properties("firstName"),
	hasLabel("comment").out("replyOf").out("hasCreator")
	.properties("firstName")
	).as("author").select("msg", "author")

IS3	g.V().has("ori_id", "ID").union(
	outE("knows").as("knowEdge1").inV().as("friend1")
	.select("knowEdge1").properties("creationDate")
	.as("creationDate1").select("friend1", "creationDate1"),
	inE("knows").as("knowEdge2").outV().as("friend2")
	.select("knowEdge2").properties("creationDate").
	as("creationDate2").select("friend2", "creationDate2")
	)

IS4	g.V().has("ori_id", "ID").union(hasLabel("comment").properties("content", "creationDate"), hasLabel("post").properties("imageFile", "creationDate"))
IC1	g.V().has("ori_id", "ID").as('a').both("knows").as('b'). both("knows").where(neq('a')).as('c'). both("knows").where(neq('a')).as('d'). union(select('b'), union(select('c'), select('d'))) .has("Key", "Value").as("person").union(union(out("isLocatedIn").properties("name"), out("studyAt").out("isLocatedIn").properties("name")), out("workAt").out("isLocatedIn").properties("name")).as("place").select("person", "place")
IC2	g.V().has("ori_id", "ID").both("knows").in("hasCreator"). has("creationDate", lte("Date")). order("creationDate", decr).limit(1).properties()
IC3	g.V().has("ori_id", "ID").as('a').union(both("knows"), both("knows").both("knows")).where(neq('a')).dedup().not(out("isLocatedIn").out("isPartOf").or(has("CountryKey", "Country1"), has("CountryKey", "Country2"))).as("person").in("hasCreator"). has("creationDate", between("StartDate", "EndDate")). or(out("isLocatedIn").has("*CountryKey*","Country1"), out("isLocatedIn").has("*CountryKey*","Country2")).select("person").groupCount()
IC2	g.V().has("ori_id", "ID").both("knows").in("hasCreator"). hasLabel("post").has("creationDate", between("StartDate", "EndDate")).out("hasTag").not(in("hasTag").hasLabel("post"). has("creationDate", lte("Date"))).groupCount("name")

.