



# The Graph Database Interface: Scaling Online Transactional and Analytical Graph Workloads to Hundreds of Thousands of Cores

Maciej Besta\*<sup>†</sup>  
Robert Gerstenberger\*  
ETH Zurich

Marc Fischer  
ETH Zurich

Michał Podstawski  
TCL Eagle Lab  
Warsaw University of Technology

Nils Blach  
ETH Zurich

Berke Egeli  
ETH Zurich

Georgy Mitenkov  
ETH Zurich

Wojciech Chlapek  
ICM UW

Marek Michalewicz  
Sano Centre for Computational  
Medicine

Hubert Niewiadomski  
Cledar

Jürgen Müller  
BASF SE

Torsten Hoefler<sup>†</sup>  
ETH Zurich

## ABSTRACT

Graph databases (GDBs) are crucial in academic and industry applications. The key challenges in developing GDBs are achieving high performance, scalability, programmability, and portability. To tackle these challenges, we harness established practices from the HPC landscape to build a system that outperforms all past GDBs presented in the literature by orders of magnitude, for both OLTP and OLAP workloads. For this, we first identify and crystallize performance-critical building blocks in the GDB design, and abstract them into a portable and programmable API specification, called the Graph Database Interface (GDI), inspired by the best practices of MPI. We then use GDI to design a GDB for distributed-memory RDMA architectures. Our implementation harnesses one-sided RDMA communication and collective operations, and it offers architecture-independent theoretical performance guarantees. The resulting design achieves extreme scales of more than a hundred thousand cores. Our work will facilitate the development of next-generation extreme-scale graph databases.

## CCS Concepts

• **Information systems** → **Graph-based database models**; *Parallel and distributed DBMSs*; *Database design and models*; Distributed database transactions; • **Computer systems organization** → Distributed architectures.

\*Both authors contributed equally to this research.

<sup>†</sup>Corresponding authors.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

SC '23, November 12–17, 2023, Denver, CO, USA

© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-0109-2/23/11...\$15.00

<https://doi.org/10.1145/3581784.3607068>

## Keywords

Graphs, Graph Databases, Data Layout, Graph Queries, Graph Transactions, Labeled Property Graph, RDMA

## ACM Reference Format:

Maciej Besta, Robert Gerstenberger, Marc Fischer, Michał Podstawski, Nils Blach, Berke Egeli, Georgy Mitenkov, Wojciech Chlapek, Marek Michalewicz, Hubert Niewiadomski, Jürgen Müller, Torsten Hoefler. 2023. The Graph Database Interface: Scaling Online Transactional and Analytical Graph Workloads to Hundreds of Thousands of Cores. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '23)*, November 12–17, 2023, Denver, CO, USA. ACM, New York, NY, USA, 14 pages. <https://doi.org/10.1145/3581784.3607068>

**Code and GDI Specification:** <https://github.com/spcl/GDI-RMA>

**Extended Technical Report:** <https://arxiv.org/abs/2305.11162>

## 1 INTRODUCTION

Graph databases (GDBs) enable storing, processing, and analyzing large and evolving irregular graph datasets in areas as different as medicine or sociology [17, 73]. GDBs face unique design and compute challenges. First, GDB datasets are huge and complex. While they can have over tens of trillions of edges [66], both vertices and edges may also come with arbitrarily many labels and properties. This further increases dataset sizes. On top of that, *much* larger datasets are already on the horizon<sup>1</sup>. Second, while traditional GDB workloads focus on online transactional processing (OLTP), there is a growing interest in supporting other classes such as online analytical processing (OLAP) or the “online serving processing” (OLSP), also called business intelligence [65]. *How to design high-performance and scalable databases that enable processing of large and complex graphs for OLTP, OLAP, and OLSP queries?* Third, portability is also important - there are many different hardware architectures available, and it may be very tedious and expensive to port a database codebase to each new class of hardware. Finally, a GDB design that would satisfy all the above challenges may become extremely complicated, and consequently hard to reason about, debug, maintain, or extend. This raises the question: *How to ensure*

<sup>1</sup>As indicated by discussions with our industry partners

Reference	RDMA?	Prog.?	Port.?	Focus on...						Achieved scales (OLTP)					Achieved scales (OLAP, OLSP)					MemS?	Th.?
				wR	bR	OLTP	OLAP	OLSP	BULK	#S	#C	Size	E	V	#S	#C	Size	E	V		
A1 [25]	☐	✗	✗	✗	✗	☐	✗	✗	✗	245	2,940	3.2 TB	6.2B	3.7B	✗	✗	✗	✗	✗	128 GB	✗
GALA [87]	✗	✗	✗	✗	✗	☐	✗	✗	✗	✗	✗	✗	✗	✗	16	384	1.96 TB	17.79B	2.69B	512 GB	✗
G-Tran [30]	☐	✗	✗	☐	✗	✗	✗	✗	✗	10	160	*1.28 TB	0.495B	0.082B	✗	✗	✗	✗	✗	128 GB	☐
Neo4j [1]	✗	✗	✗	☐	☐	☐	☐	☐	☐	1	128	6.9 TB	55B	5B	1	128	6.9 TB	55B	5B	2 TB	✗
TigerGraph [100]	✗	✗	✗	✗	✗	☐	✗	☐	☐	40	1600	17.7 TB	533.5B	72.62B	36	4,608	N/A	539.6B	72.6B	1 TB	✗
JanusGraph [67]	✗	✗	✗	☐	☐	☐	☐	☐	☐	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	✗
Weaver [36]	✗	✗	✗	☐	☐	☐	☐	☐	☐	44	352	0.976 TB	1.2B	0.08B	✗	✗	✗	✗	✗	16 GB	✗
Wukong [93]	☐	✗	✗	✗	✗	☐	✗	✗	✗	6	120	*0.384 TB	1.41B	0.387B	✗	✗	✗	✗	✗	64 GB	✗
ByteGraph [65]	✗	✗	✗	☐	☐	☐	☐	☐	☐	10	160	N/A	N/A	N/A	130	N/A	113 TB	N/A	N/A	1 TB	☐
<b>This work</b>	☐	☐	☐	☐	☐	☐	☐	☐	☐	<b>7,142</b>	<b>121,680</b>	<b>77.3 TB</b>	<b>549.8B</b>	<b>34.36B</b>	<b>7,142</b>	<b>121,680</b>	<b>36.5 TB</b>	<b>274.9B</b>	<b>17.2B</b>	<b>64 GB</b>	☐

**Table 1:** Comparison of graph databases. “RDMA?”: Is a system primarily targeting RDMA architectures? “Prog.?”: Does a system’s storage and transactional backend design focus on programmability and code simplicity? “Port.?”: Does a system foster portability? If yes, is it portability within different RDMA architectures (“wR”), or also beyond RDMA (“bR”) “Supported workloads”: What are supported workloads? (all workloads are explained in Section 4) “Achieved scales”: What are achieved scales? “#S”: Number of servers. “#C”: Number of cores. “Size”: Total size (in memory) of the processed graph. “|E|”: Number of edges. “|V|”: Number of vertices. “MemS”: Amount of memory available in a single server. “Th.?”: Does a system come with theoretical analysis and support for its performance and scalability properties? “☐”: full support, “☐”: partial support, “✗”: no support, “\*”: Estimate. *The GDI-based system is the only one that focuses on all major aspects of the GDB design: programmability, portability, high performance, and very large scales.*

portability and programmability of complex next-generation graph databases, without compromising on their performance?

To resolve all the above challenges, we provide the first principled approach for designing and implementing large-scale GDBs. This approach harnesses some of the most powerful practices and schemes from the HPC domain, several of them for the first time in GDB system design. Our approach is inspired by the Message-Passing Interface (MPI) [75] and numerous successes it has in designing and developing portable, programmable, high-performance, and scalable applications. We propose to approach the GDB design in a similar way: (1) identify performance-critical building blocks, (2) build a portable API, (3) implement this API with high-performance techniques such as collectives or one-sided RDMA, and (4) use the API implementation to build the desired GDB system. In this work, we execute these four steps, and as a result we deliver a publicly-available GDB system that resolves all the four challenges.

First, we analyze the design and codebases of many GDBs [17] (e.g., Neo4j [90], Apache TinkerPop [8], or JanusGraph [67]) to identify fundamental performance-critical building blocks. We then crystallize these blocks into a portable and programmable specification called the Graph Database Interface (GDI) (contribution #1). GDI focuses on the data storage layer, covering database transactions, indexes, graph data, graph metadata, and others. GDI is portable because – as MPI – it is fully decoupled from its implementation. Hence – just like with MPI-based applications – any database based on GDI could be seamlessly compiled and executed on any system, if there is a GDI implementation for that system.

Second, we offer a high-performance implementation of GDI for distributed-memory (DM) systems supporting RDMA-enabled interconnects, called GDI-RMA. We use GDI-RMA to build a highly-scalable GDB engine (contribution #2). We focus on DM systems as they enable keeping data fully in-memory to avoid expensive disk accesses. Simultaneously, RDMA has been the enabler of scalability and high performance in both the supercomputing landscape and – more recently – in the cloud data center domain [42, 55, 94, 102].

In GDI-RMA, we make three underlying design decisions for highest performance and scalability. First, we carefully design a scalable distributed storage layer called blocked graph data layout (BGDL) to enable a tradeoff between the needed communication and storage. Second, we incorporate the highly scalable one-sided non-blocking RDMA communication (puts, gets, and atomics). Third, we use collective communication (collectives) [28, 51] to deliver scalable transactions involving many processes (e.g., for large-scale OLAP queries) with well-defined semantics.

We support nearly any function in our implementation with a theoretical performance analysis that is independent of the underlying hardware (contribution #3). This facilitates the reasoning about the performance and scalability of our GDI implementation.

We illustrate how to use GDI to program many graph database workloads (contribution #4), covering OLTP, OLAP, and OLSP. We consider recommendations by the LDBC and LinkBench academic and industry benchmarks [5, 12]. We use established problems such as BFS [18, 82] and state-of-the-art workloads such as Graph Neural Networks [16, 24, 43, 46, 59, 91, 103, 105, 106]. Moreover, as there are no publicly available graph datasets with labels and properties of that magnitude, we also develop an in-memory distributed generator that can rapidly create such a graph of arbitrary size and configuration of labels and properties (contribution #5).

The evaluation of GDI-RMA (contribution #6) significantly surpasses in scale previous GDB analyses in the literature in the counts of servers, counts of cores, and in the size of a single analytic workload (see Table 1). We successfully scale to 121,680 cores (7,142 servers), using all the available memory, and the only reason why we did not try more is because we do not have access to a larger system. Based on our analysis, we expect that our GDI implementation could easily achieve the scale of hundreds of thousands of cores. We also achieve high throughput and low latencies over databases such as Neo4j or JanusGraph by more than an order of magnitude in both metrics. Our implementation is publicly available (contribution #7) to help achieve new frontiers for GDBs running on petascale and exascale data centers and supercomputers.

We compare our work to other GDBs in Table 1. GDI is the only RDMA-based system to support all three fundamental workloads (OLTP, OLAP, OLSP), and the only one to focus on portability & programmability, and with theoretical performance guarantees.

## 2 GRAPH DATA MODEL & WORKLOADS

We first present basic concepts and notation.

*Graph Data Model* We target graphs modeled with the established Labeled Property Graph Model (LPG) [17], a primary data model used in many GDBs, including the leading Neo4j GDB [17]. An LPG graph can formally be modeled as a tuple  $(V, E, L, l, K, W, p)$ .  $V$  is a set of vertices and  $E \subseteq V \times V$  is a set of edges;  $|V| = n$  and  $|E| = m$ .  $L$  denotes the set of labels that differentiate subsets of vertices and edges.  $l$  is a labeling function, which maps vertices and edges to subsets of labels;  $l : V \cup E \mapsto \mathcal{P}(L)$  with  $\mathcal{P}(L)$  being the power set of  $L$ , meaning all possible subsets of  $L$ . Each vertex and edge can also feature arbitrarily many properties (sometimes

referred to as attributes). A property is a  $(key, value)$  pair, where the  $key$  works as an identifier with  $value$  is the corresponding value.  $K$  and  $W$  are sets of all possible keys and values, respectively.  $p : (V \cup E) \times K \mapsto W$  maps each vertex and edge to their properties, given the key. We also refer to the elements of  $K$  as *property types* ( $p$ -types). Note that we distinguish between property types and *properties*, the latter being the specific key-value property tuples attached to vertices/edges.

**Graph Data vs. Graph Metadata** We collectively denote labels  $L$ , property types  $K$  and property values  $W$  as the **graph metadata** because these sets do not describe any specific graph elements, but they define the potential labels, keys, and values. A collective name **graph data** refers to the actual graph elements, described by  $V, E, l, p$ .

**Graph Database Workloads** The established LDBC and LinkBench academic and industry benchmarks [5, 12] identify two main classes of graph database workloads targeting the LPG graph model: *interactive workloads* [37] (mostly OLTP) and *graph analytics* [53] (mostly OLAP). Interactive workloads are further divided into *short read-only queries* (which often start with a single graph element such as a vertex and lookup its neighbors or conduct small traversals) and *transactional updates* (which conduct simple graph updates, such as inserting an edge). Next, preliminary efforts also distinguish an additional class of *business intelligence workloads* (BI) [96, 97] which fetch large parts of a graph and often use data summarization and aggregation. They are sometimes referred to as *Online Serving Processing* (OLSP) [65]. Finally, we also distinguish workloads associated with bulk data ingestion (BULK). They take place, e.g., when inserting new batches of data into the system.

### 3 THE GRAPH DATABASE INTERFACE

GDI is a storage layer interface for GDBs, offering CRUD (create, read, update, delete) functionality for the elements of the LPG model: vertices, edges, labels, and properties. The interface provides rich semantics and transaction handling. The focus of GDI lies on enabling high-performance, scalable, and portable implementations of the provided methods. Moreover, GDI facilitates programmability by offering a structured set of routines with well-defined semantics.

In this paper, we provide a comprehensive summary of the most important aspects of GDI. We also distill the key design choices and insights beyond the common system knowledge, that we indicate with the “🔗” symbol. The full **GDI specification** is available in a separate manuscript [14]. It contains a detailed description of routines, extensive advice for users and implementors, naming conventions, description of basic datatypes, and others.

#### 3.1 Relation Between GDI and Graph Databases

We illustrate the relation between GDI and a generic graph database landscape in Figure 1. GDI is to be used primarily by the database middle layer, as a storage and transactional engine. Here, the **client** first queries the GDB using a graph query language such as Cypher [45]. Second, the **database mid-layer** coordinates the execution of the client query. This could include distributing the workload among multiple machines, or aggregating as well as filtering intermediate results that ran on different processes. The mid-layer relies on the underlying **storage and transaction engine**, where GDI resides. This part accesses the graph data and

translates from generic graph-related objects needed by queries to hardware dependent storage. Therefore, the layer provides a rich set of interfaces to create, read, update, and delete (CRUD) vertices, edges, labels, and properties, and to execute transactions. Finally, the **storage backend** provides access to the actual storage such as distributed RAM, using formats such as CSV files or JSON.

We also envision that GDI could be used directly by a client, to directly implement a given query. For this, we will illustrate how to implement different GDB workloads with GDI in Section 4.

#### 3.2 Structure and Functionalities of GDI

The GDI interface is structured into groups of routines, detailed in Figure 2. General GDI and database management schemes (**gray color**) perform setup needed for any other GDI functions to be able to run. Graph metadata routines (**cyan color**) enable creating, updating, deleting, and querying different aspects of labels and property types. Graph data routines (**blue color**) provide CRUD capabilities for vertices and edges, also including adding, removing, updating, and querying labels and properties of specific vertices and edges, and bulk data loading. Transaction routines (**green color**) enable transactional processing of graph data. Indexes (**orange color**) provide indexing structures for vertices and edges, speeding up different queries. Indexes heavily use constraints to provide indexing for vertices/edges satisfying specific conditions. Routines for constraints are indicated with **brick red color**. Finally, all groups of routines heavily use error codes (**red color**) and schemes for basic datatypes (**gray color**). We now elaborate on key GDI parts.

There are two classes of GDI routines: **collective** (“[C]”) and **local** (“[L]”). All processes actively participate in a collective routine (i.e., they all explicitly call this routine), while only a single process actively participates in a local routine (it can still passively involve arbitrarily many other processes by accessing their memories). Collective communication has heavily been used in high-performance computing [28, 51]. Such communication routines, by actively involving all participating processes, are more efficient than routines based on point-to-point communication by facilitating various optimizations and advanced communication algorithms [51, 52]. They also foster portability and programmability [52] by coming with well-defined semantics for the behavior of groups of processes.

#### 3.3 High-Performance Transactions

A transaction consists of a sequence of operations on graph data, and it must guarantee Atomicity, Consistency, Isolation, and Durability (ACID). GDI poses no restriction on how to ensure ACID. GDI

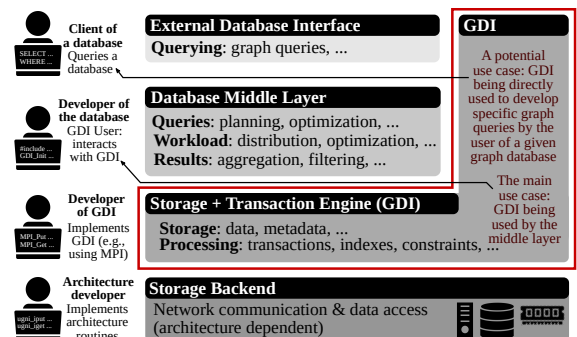


Figure 1: GDI with respect to other parts of the graph database landscape.

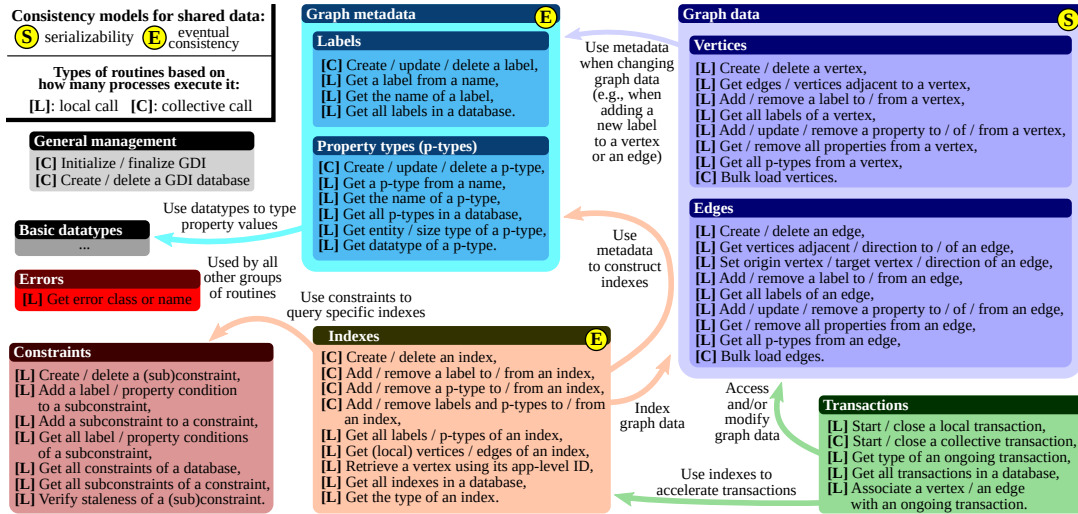


Figure 2: Illustration of the classes of GDI routines.

transactions support full CRUD functionality for vertices, edges, and their associated labels and properties. Accessing and modifying graph data is conducted only within a transaction body. Any single process can be in arbitrarily many concurrent transactions.

**Local (single process) transactions** are transactions that a single process has started. This type of a transaction is meant for graph operations which touch only a small part of the graph. **Collective transactions** are transactions which actively involve all processes; they are used to execute large OLAP or OLSP queries.

🔑 **Major Design Choice & Insight:** Use collective transactions, that involve all processes, for global OLAP/OLSP workloads. This facilitates not only low latencies (as collectives are highly tuned) but also programmability (as collectives have well-defined semantics).

GDI also distinguishes **read transactions** from **write transactions**. This further facilitates high-performance implementations, by providing opportunities for optimized read-only transactions that can assume that no participating process modifies the data.

### 3.4 Fast & Effective Access to Graph Data

GDI provides fast transactional access to vertices, and their labels and properties, with a two-step scheme. In the first step, an application-level vertex ID is translated to an internal GDI-specific ID. This makes GDI more portable, as it is independent of any details of how the higher-level system layers may implement IDs. In our implementation, we use internal indexing structures for this translation. This internal ID uniquely identifies a vertex in the whole GDI database.

GDI offers two types of internal IDs: *volatile* and *permanent*. The former are valid only during the transaction within which they are obtained. This facilitates optimizations such as the dynamic relocation of graph data, but it also requires re-obtaining these IDs in each transaction. The latter are shared across transactions, which reduces the number of remote operations, but hinders dynamic load balancing. The user can choose the most suitable variant.

🔑 **Major Design Choice & Insight:** Volatile IDs facilitate optimizations related to load balancing. For example, it facilitates redistributing the graph across processes between collective transactions, without fearing that internal IDs become stale.

### 3.5 Handles

Internal representations of objects involved in transactions, such as vertices or property types, are not directly accessible to the GDI user. To enable fast and programmable way of accessing and manipulating graph data within transactions, GDI prescribes using **handles (access objects)**, i.e., opaque objects that hide the internal implementation details of accessed objects, and represent these objects *on the executing process*. To create a handle for an existing vertex  $v$  or edge  $e$ , the user calls `GDI_AssociateVertex( $v$ )` or `GDI_AssociateEdge( $e$ )`;  $v$  and  $e$  are respective internal IDs.

🔑 **Major Design Choice & Insight:** Using handles to access opaque objects improves usability. First, it enables the GDI implementation to decide on the details of how graph data is accessed. Using a handle enables remote direct (zero-copy) memory access, but it could also be used to transparently copy or move the data, for example for dynamic relocation or to cache the data locally. Moreover, it relieves the user of ensuring that there are no pending operations involving out-of-scope, opaque objects; the GDI implementation instead takes care of that. It also allows users to simply mark objects for deallocation, relying on the GDI implementation to retain the object until all pending operations have completed. Requiring handles to support native-language assignment and comparison operations keep the GDI interface clean and simple.

### 3.6 Consistency

For performance reasons, GDI enables different consistency models. The interface requires **serializability** for graph data (vertices, edges, and their associated labels and properties). Generally, this data can only be altered by transactions that ensure ACID. Second, GDI guarantees **eventual consistency** for metadata (labels, property types) and for indexes. Since these objects also affect the graph data, this might lead to cases where graph data becomes inconsistent until the system has converged. Transactions must be able to detect such state and abort accordingly. Note that implementations are free to provide consistency models for metadata and for indexes that are more restrictive (stronger) than eventual consistency.

🔑 **Major Design Choice & Insight:** Enabling separate consistency models for data and metadata fosters flexibility and simplicity.

Many systems only specify their compliance with the Consistency requirement of ACID, but do not clearly define what type of consistency they employ [90]. In GDI, we clearly specify it.

**Major Design Choice & Insight:** *Clearly specifying the used consistency model fosters programmability.*

## 4 GRAPH WORKLOADS WITH GDI

We now illustrate how to use GDI to easily and portably implement representative queries from all major classes of GDB workloads. In principle, one could implement all of these workloads with single-process transactions. In GDI, we observe that, for some of these workloads, if they harness all processes in a database, this gives more performance. Thus, it is often more beneficial to use collective transactions in such cases. We summarize what types of transactions are best to be used for what workloads in Table 2.

Workload class	Type	Best-suited GDI routines
Interactive (short)	read-only	OLTP Single-process transactions
Interactive (complex)	read-only	OLTP Single-process transactions
Interactive (updates)	read/write	OLTP Single-process transactions
Graph analytics	read-only	OLAP Collective transactions
Business intelligence	read-only	OLSP Single-process or collective trans.
Massive data ingestion	read/write	BULK Bulk data loading collectives

**Table 2: Key graph database workloads (see Section 2 for details) and the associated recommended mechanisms of GDI best used for implementation.**

Listings 1, 2, and 3 contain – respectively – a simple OLTP interactive query (fetching properties from a small vertex set), an OLAP query (a convolutional Graph Neural Network (GNN)), and an OLSP transaction. For clarity, we omit straightforward additions (e.g., error handling or checking if transactions fail). In all the queries, for each accessed vertex or edge, one first translates the application-level ID to the GDI ID, and then uses the obtained ID to create handles to be able to access the corresponding graph data. The used symbols are as follows: `trans_obj` (a handle to the state of the ongoing transaction), `vH` (a handle to a vertex  $v$ ), `eH` (a handle to an edge  $e$ ), `vID` (an internal GDI ID for a vertex  $v$ ), `vID_app` (an external application-level ID for a vertex  $v$ ).

```

1 GDI_StartTransaction(&trans_obj);
2 GDI_TranslateVertexID(&vID, GDI_LABEL_PERSON, &vID_app,
   trans_obj); //Find internal vertex ID (vID) based on the
   application-level ID (vID_app)
3 GDI_AssociateVertex(vID, trans_obj, &vH); //Create a temporary
   access object for vertex vID
4 GDI_GetEdgesOfVertex(&eIDs, GDI_EDGE_UNDIRECTED, vH); //Retrieve
   all undirected edges
5 for each eID in eIDs do {
6   GDI_AssociateEdge(eID, trans_obj, &eH); //Create a temporary
   access object for edge eID
7   GDI_GetAllLabelsOfEdge(&labels, eH);
8   if(/* one of the labels equals GDI_LABEL_FRIENDDOF */) {
9     GDI_GetVerticesOfEdge(&v_originID, &v_targetID, eH); //
       Retrieve target vertex
10    neighborsID.add(v_targetID) /* add target vertex to
       neighborsID data structure. Details of neighborsID are
       omitted for clarity */ } }
11 for each vID in neighborIDs do {
12   GDI_AssociateVertex(vID, trans_obj, &vH);
13   GDI_GetPropertiesOfVertex(&fName, GDI_PROP_TYPE_FNAME, vH);
14   GDI_GetPropertiesOfVertex(&lName, GDI_PROP_TYPE_LNAME, vH);
15   /* add fName, lName to the data structure to be returned */ }
16 GDI_CloseTransaction(&trans_obj);

```

**Listing 1:** C-style pseudocode of an example interactive OLTP query with GDI. Here, we retrieve the first and last name of all persons that a given person, modeled with a vertex `vID_app`, is friends with. For this, we first obtain all edges of `vID_app` (line 4), iterate over them to find edges corresponding to friendships (lines 5-10), preserve the corresponding neighbors (line 10), and retrieve the names and surnames of each such neighbor (lines 11-15).

```

1 for(l = 0; l < layers /* a user parameter */; ++l) {
2   /* some form of collective synchronization */
3   GDI_StartTransaction(&trans_obj);
4   GDI_GetLocalVerticesOfIndex(&vIDs, v_index, trans_obj); //
       Retrieve local vertices
5   for each vID in vIDs do {
6     GDI_AssociateVertex(vID, trans_obj, &vH);
7     GDI_GetPropertiesOfVertex(&feature_vec,
       GDI_PROP_TYPE_FEATURE_VEC, vH); //Get the vertex feature
       vector stored as a property
8     GDI_GetNeighborVerticesOfVertex(&nIDs, GDI_EDGE_OUTGOING, vH);
       //Retrieve neighborhood vertices
9     for each nID in nIDs do {
10      GDI_AssociateVertex(nID, trans_obj, &nH);
11      GDI_GetPropertiesOfVertex(&feature_vec_n,
       GDI_PROP_TYPE_FEATURE_VEC, nH);
12      feature_vec += feature_vec_n; /* Apply the aggregation GNN
       phase; in this example, we use a summation */ }
13      feature_vec = MLP(feature_vec); //Apply the update GNN phase;
       in this example, we use a simple MLP transformation
       defined externally by the user
14      feature_vec = sigma(feature_vec); //Apply the non-linearity
       defined by the user
15      GDI_UpdatePropertyOfVertex(&feature_vec,
       GDI_PROP_TYPE_FEATURE_VEC, vH); }
16 GDI_CloseTransaction(&trans_obj); }

```

**Listing 2:** C-style pseudocode of an example OLAP query with GDI (graph convolution network training/inference). The details of graph convolution are beyond the scope of this work and they can be discussed in detail in rich existing literature [16, 103]. In brief, this query consists of a specified number of iterations (“layers”). In each layer, every vertex first updates itself based on the features of its neighbors (“aggregation”, lines 9-12) and then the outcomes are processed by a multilayer perceptron (MLP, line 13) and a non-linearity (line 14). Finally, the property modeling the feature vector of each vertex is updated accordingly (line 15). Due to space constraints, we present a simplified query with the most important communication-intensive operations; a full version with all other parts such as weight updates is in the extended technical report.

```

1 local_count = 0;
2 GDI_StartCollectiveTransaction(&trans_obj);
3 //Index_obj indexes all vertices with label GDI_LABEL_PERSON
4 GDI_GetLocalVerticesOfIndex(&vIDs, index_obj, trans_obj);
5 for each person in vIDs do {
6   GDI_AssociateVertex(person, trans_obj, &vH);
7   GDI_GetPropertiesOfVertex(&age, GDI_PROP_TYPE_AGE, vH);
8   if(age <= 30) { continue; } //The condition is not met
9   /* Define a constraint "cnstr" with a label condition "=="
       GDI_LABEL_OWN" (to check for the act of owning) */
10  GDI_GetNeighborVerticesOfVertex(&things, cnstr,
       GDI_EDGE_OUTGOING, vH); //Get neighbors satisfying cnstr
11  for each object in things do {
12    GDI_AssociateVertex(object, trans_obj, &vH);
13    GDI_GetAllLabelsOfVertex(&labels, vH);
14    if(/* no label equals GDI_LABEL_CAR */) { continue; }
15    GDI_GetPropertiesOfVertex(&color, GDI_PROP_TYPE_COLOR, vH);
16    if(color == red) { local_count++; } } }
17 GDI_CloseCollectiveTransaction(&trans_obj);
18 reduce(local_count);

```

**Listing 3:** C-style pseudocode of an example business intelligence workload with GDI: “MATCH (per:Person) WHERE per.age>30 AND per:OWN->vehicle(:Car) AND vehicle.color equals red RETURN count(per)”. Here, we first fetch all vertices modeling people (using an index, line 4), check whether each such person satisfies the specified criteria (lines 5-16), including age (lines 7-8), car ownership (lines 9-14), and the car color (lines 15-16).

## 5 SCALABLE GDI RDMA IMPLEMENTATION

Our high-performance implementation of GDI, called GDI-RMA (GDA), is based on MPI and it uses RDMA-enabled one-sided communication as the high performance and high scalability driver. While there are different ways to harness RDMA, for **highest performance**, we focus on **one-sided fully-offloaded communication**. Here, processes communicate by directly accessing dedicated portions of one another’s memories called a *window*. Communication bypasses the OS and the CPU, eliminating different overheads. Such accesses are conducted with *puts* and *gets* that – respectively – write to and read from remote memories. *Puts/gets* offer very low latencies, often outperforming message passing [42]. One can also use remote *atomics* [15, 49, 75, 92]; here, we additionally harness

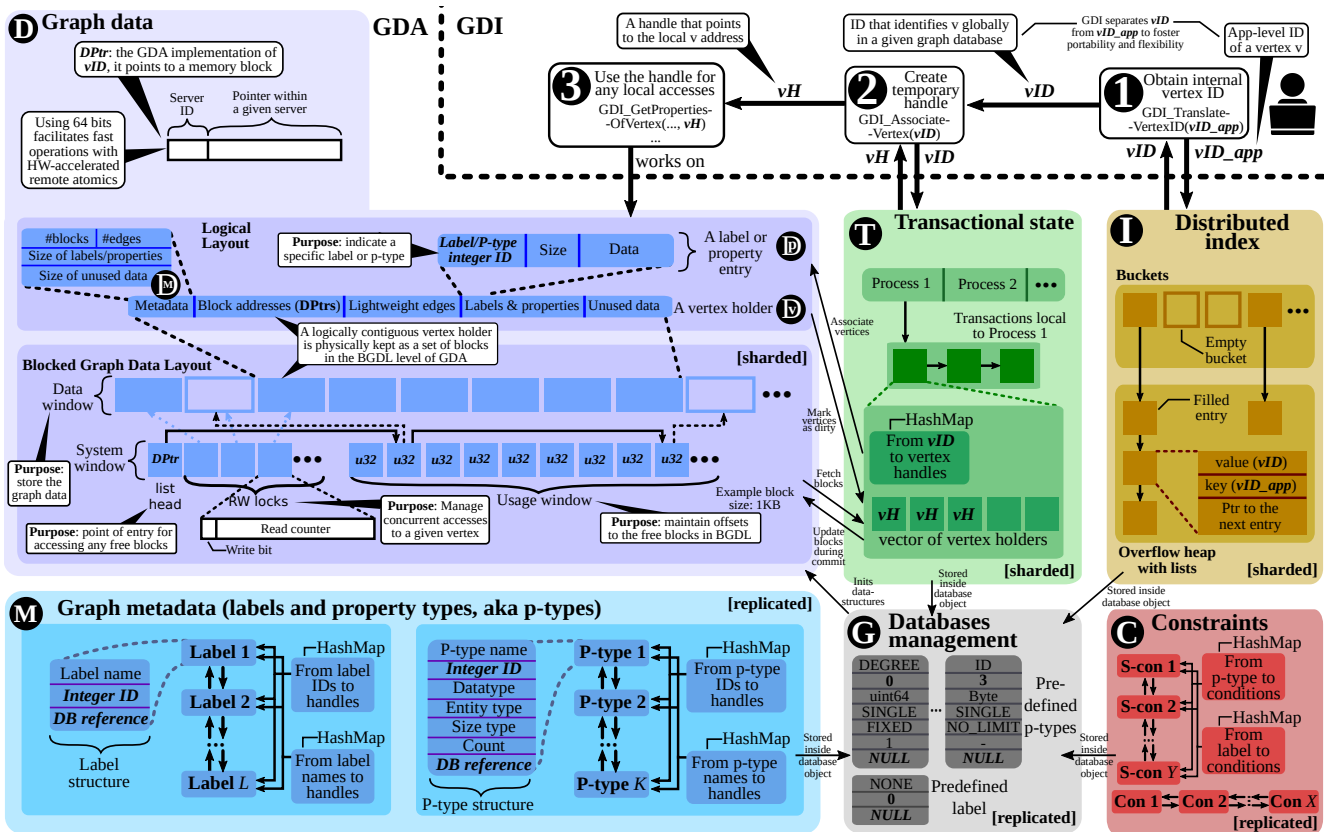


Figure 3: Details of the GDI-RMA (GDA) implementation, and its interaction with GDI. In the upper part of the figure, we illustrate a simple sequence of steps (taken within a transaction) to access a selected property of a given vertex  $v$ .

**hardware support for atomics** offered by RDMA networks for very fast fine-grained synchronization. For data consistency, we use *flushes* to explicitly synchronize memories. We use **non-blocking** variants of all functions, because they can additionally increase performance by overlapping communication and computation [42]. All these routines are supported by virtually any RDMA architecture, facilitating a wide portability of GDA.

We now **overview** GDA, see Figure 3. We will detail the concepts mentioned here in the following subsections. Our implementation<sup>2</sup> is fully in-memory for highest performance. GDA consists of several modules which largely correspond to the GDI classes of routines, cf. Figure 2 (we use the same color code for both illustrations). The most important modules are management structures for GDI and parallel databases (G), metadata structures (M), indexes and associated constraints (I, C), state of collective and local transactions (T) and graph data (D). Structures that are small or independent of #vertices and #edges (G, M, C) are replicated on each process to foster simplicity and high performance. All other structures are sharded<sup>3</sup>. This, combined with our fully distributed transactions, enables fast and scalable processing of very large graph datasets, being limited only by the cluster size.

## 5.1 Graph Data

The central part of GDA is associated with graph data (D in Figure 3). To make this part more manageable and programmable, it is divided into two conceptual levels. First, the **Logical Layout (LL)** level maintains structures that reflect graph data (vertices, edges, labels, properties). Importantly, these structures have *flexible sizes determined by the sizes of the corresponding parts of graph data*. For example, two vertices having different sets of labels and properties would be maintained by two structures of potentially different sizes. The LL level simplifies working with GDI from the graph developer perspective, because it enables a data-driven memory layout. However, it is challenging to operate on such variable sized and dynamic structures in an RDMA environment using one-sided communication. For this, we also provide the underlying **Blocked Graph Data Layout (BGDL)** level. BGDL maintains a large DM memory pool divided into same-sized memory blocks (tunable by the user). The purpose of BGDL is to translate the highly diverse structures from the LL level into these blocks. The memory blocks associated with one vertex/edge do not have to be stored continuously, and might not even be located on the same process or server. Such blocking enables a simple, effective, and flexible DM memory management: any data access or manipulation routines operate on same-sized blocks, and the difference between processing different parts of the graph is only in the counts of the associated blocks.

**Major Design Choice & Insight:** *Introducing and separating the LL routines from the BGDL fosters programmability. The*

<sup>2</sup>We use the foMPI implementation of MPI One-Sided routines [42].

<sup>3</sup>Partitioning of data onto separate servers.

*LL routines form a clean and graph-centric API; any performance optimizations can be done under the hood at the BGDG level.*

🔑 **Major Design Choice & Insight:** *Using fixed-size blocks in BGDG does not only simplify the design, but it also fosters higher performance.* This is because one only needs a single remote operation to fetch the data of a vertex that fits in one block.

Any access to the graph data begins with the user providing an application ID (*vID\_app*), which is then translated to the internal ID *vID* that uniquely identifies a given object in the whole database, and can be shared by multiple processes. This is conducted using the internal index (🔑). In GDA, the internal ID is implemented as a 64-bit distributed hierarchical pointer (*DPtr*). Its first 16 bits indicate the compute server, and the remaining 48 bits points to a local memory offset of the primary block of *v*. We use 64 bits to facilitate using HW accelerated remote atomics, which frequently operate on 64-bit words [42]. Then, *vID* is used to construct a handle *vH*, which is a pointer to *v* in the local memory of a calling process.

🔑 **Major Design Choice & Insight:** *Using 64-bit distributed pointers facilitates harnessing hardware accelerated remote atomic operations, which are commonly provided by different vendors.*

## 5.2 Logical Layout Level

We shard the graph data (vertices, edges, and their associated labels and properties) across all processes. We implemented 1D (vertex-based) and 2D (edge-based) graph partitioning, and use round-robin distribution (we tried other distribution schemes, they only negligibly impact our performance). GDI's specification is on purpose orthogonal to the partitioning/distribution, so it is usable with any such scheme.

*Vertices & Edges* The data structure of each vertex *v* or an edge *e* (called a *vertex* or *edge holder*, see 📦) is divided into, respectively, metadata (selected important information used for the data management, see 📦), block addresses (addresses of blocks that store the data), lightweight edges (*v*'s edges that do not contain many labels or properties and are thus stored together with *v* for more performance), the label and property data (see 📦), and any unused memory.

*Lightweight Edges* Many GDB queries involve iterating over edges of a given vertex. Simultaneously, in many graph datasets, only vertices have rich additional data (labels, properties), while edges often do not carry such data (e.g., in many citation networks). To maximize performance for these cases, we introduce *lightweight edges* in GDA. Each such edge has at most one label. Importantly, these edges are stored in the vertex holder object of their source vertex. This enables very fast access.

## 5.3 Transactions & ACI

Each transaction is represented by a state with any necessary information (e.g., which dirty blocks must be written back into the distributed graph storage when the transaction commits). By combining hashmaps and linked lists for keeping track of any blocks used within a transaction, we achieve highly efficient transactions where any operation on a block is done in  $O(1)$  time.

GDA uses a two-phase scalable reader-writer (RW) locking to ensure the ACI properties. Only one lock per any vertex *v* is used to reduce the number of remote atomics. Figure 3 shows the lock data structure, located in the system window at a corresponding offset

to the primary block of *v*'s holder object. The *write bit* determines if a process holds a write lock to *v*, while the *read counter* indicates the number of processes that currently hold a read lock on *v*.

🔑 **Major Design Choice & Insight:** *Fully offloaded RDMA design facilitates high performance. While being complex, it is kept under the hood and does not adversely impact GDI's programmability.*

## 5.4 Graph Metadata

We replicate graph metadata on each process for performance reasons. This is because both *L* and *P* are in practice much smaller than *n*. A label is represented by a structure that holds a label name, an integer ID, and a reference to the associated graph database. A property structure is similar, with the difference that it contains additional information (the entity type, the GDI datatype, the size type, and the size limitation). We summarize these structures in Figure 3 (📦). When storing specific labels and properties on vertices/edges, we only use their associated integer IDs. To enable fast accesses to graph metadata, we maintain double linked-lists of labels and properties, as well as hash maps. The former enables to add and remove labels or properties in  $O(1)$  work (given the handle), the latter is used for checking their existence in  $O(1)$ .

🔑 **Major Design Choice & Insight:** *Replicating metadata simplifies the design without significantly increasing the needed storage.*

## 5.5 Summary of Parallel Performance Analysis

Each routine in GDA is supported with theoretical analysis of its performance, in order to ensure GDA's *performance portability*. For this, we use the **work-depth (WD) analysis**. Intuitively, the *work* of a given GDA routine is the total number of operations in the execution of this routine, and the *depth* is the longest sequential chain of dependencies in this execution [19, 21].

Due to space constraints, we provide the work and depth of GDA routines in a full extended version of the GDA manuscript. Importantly, the majority of GDA routines (both for data and metadata management) come with *constant*  $O(1)$  *work and depth*. Only a few routines that modify *x* metadata items (property types or labels) come with  $O(x)$  work and depth. This also implies *low overheads in practice*, as *x* is usually a small number (i.e., fewer than 10-20).

## 6 EVALUATION

We now illustrate how GDI and its implementation GDA ensure high performance (latency, throughput) and large scale.

### 6.1 Experimental Setup, Workloads, Metrics

We first sketch the evaluation methodology. For measurements, we omit the first 1% of performance data as warmup. We derive enough data for the mean and 95% non-parametric confidence intervals. We use arithmetic means as summaries [50].

As **computing architectures**, we use the Piz Daint Cray super-computer installed in the Swiss National Supercomputing Center (CSCS). Piz Daint hosts 1,813 XC40 and 5,704 XC50 servers. Each XC40 server has two Intel Xeon E5-2695 v4 @2.10GHz CPUs (2x18 cores, and 64 GB RAM). Each XC50 server comes with a single 12-core Intel Xeon E5-2690 HT-enabled CPU, and 64 GB RAM). The interconnect between servers is Cray's Aries based on the Dragonfly topology [38, 58]. We use **full parallelism**, i.e., we run algorithms on the maximum number of cores available.

We consider three **metrics**: **latency** (i.e., how fast a query finishes), **throughput** (i.e., how many queries can we execute per time unit), and **scale**. For scale, we (1) increase the number of servers *together with* the size of the dataset (the so called “**weak scaling**”) and (2) increasing the number of servers *for a fixed* dataset (the so called “**strong scaling**”).

## 6.2 Selecting Baselines and Related Challenges

While there exist many graph databases, the vast majority of them is not freely available. We attempted to get access to different systems, such as Oracle’s PGX, but our attempts were unsuccessful. Among the available systems, we shortlisted databases that provide full support for both OLTP and OLAP queries. After an extensive investigation and configuration effort, we were able to successfully configure and use Neo4j (5.10) [90] and JanusGraph (0.6.2) [67]. We configure both baselines for in-memory execution. Additionally, to maximize the performance of comparison baselines, we use their high-performance consistency guarantees (e.g., eventual consistency for JanusGraph), even if GDI provides serializability for graph updates. *These two systems are two of the highest-ranking core graph databases (i.e., systems with the database model “Graph”) in the DB-Engines Ranking.*

## 6.3 Distributed In-Memory LPG Graph Generator for Massive-Scale Experiments

Obtaining appropriate graph datasets is challenging due to the fact that we target graphs of very large scales *and* having rich amounts of labels and properties. Existing generators experienced regular OOM problems when using large scales, while available real-world graphs have no labels/properties and are not large enough. Hence, to facilitate large-scale graph database experiments, we develop an *in-memory distributed generator of LPG graphs* that enables *fast* construction of *arbitrarily large LPG datasets limited only by the available compute resources*, fully in-memory, so that they are immediately available for further processing. We base our generator on the existing code provided by the Graph500 benchmark [76] that uses the realistic Kronecker random graph model with a heavy-tail skewed degree distribution [64]. We extend this model by adding support for a user-specified selection (i.e., counts and sizes) of labels and properties, and how they are assigned to vertices and edges. By default, we use 20 different labels and 13 property types in the following analyses (we also experiment with varying these values).

## 6.4 Analysis of OLTP Workloads

We first analyze the OLTP workloads. Here, we stress GDA with a high-velocity stream of graph queries and transactions. We use four specific scenarios based on the LinkBench benchmark [12] and on other past GDB evaluations [30, 36], see Table 3 for details.

We first evaluate the overall throughput, see Figure 4. GDA achieves high scalability: adding more servers consistently improves the throughput in both strong and weak scaling. Throughput increase is particularly visible in the RI and RM workloads with more read queries, because LB and WI workloads come with more updates that involve more synchronization and communication. We also observe that, overall, XC50 servers give more performance than XC40, especially for RM workloads dominated by reads. We conjecture this is due to the XC50 servers offering more network bandwidth per core. Moreover, we note that very low percentages

Operation	“Read Mostly” (RM) [36]	“Read Intensive” (RI) [36]	“Write Intensive” (WI) [30]	LinkBench (LB) [12]
<b>Read queries:</b>	<b>99.8%</b>	<b>75%</b>	<b>20%</b>	<b>69%</b>
Get vertex properties	28.8%	21.7%	9.1%	12.9%
Count edges of a vertex	11.7%	8.8%	0%	4.9%
Get edges of a vertex	59.3%	44.5%	10.9%	51.2%
<b>Update queries:</b>	<b>0.2%</b>	<b>25%</b>	<b>80%</b>	<b>31%</b>
Add a new vertex	0%	0%	20%	2.6%
Delete a vertex	0%	0%	6.7%	1%
Update a vertex property	0%	0%	13.3%	7.4%
Add a new edge	0.2%	25%	40%	20%

**Table 3: OLTP workloads described in this paper. We varied the fractions of specific types of operations for broad investigation beyond the ones provided here, all results followed similar patterns to those described here.**

of failed transactions (less than 0.2% for RI/RW and less than 2% for LB/WI) across all benchmarks indicate GDA’s capability to successfully resolve a sustained stream of incoming user requests, even at very high scales. Overall, the results indicate that GDA is able to both accelerate requests into a given fixed dataset (as seen by the throughput increase in strong scaling) as well as it enables scaling to larger datasets (as indicated by the throughput increase in weak scaling).

We also show histograms of latencies of different operations within a given OLTP workload. Figure 5 shows the data for LB (we plot separate latencies for transactions running on 1–8 servers). GDA is consistently the fastest, with the vast majority of its operations being below  $1\mu\text{s}$  (for 1 server) and close to  $10\text{--}100\mu\text{s}$  (for more servers), even for demanding vertex deletions. JanusGraph requires at least  $500\mu\text{s}$  for all the operations (in most of cases), with no operation being faster than  $200\mu\text{s}$ , even for the single server scenario. Vertex deletions start at around  $2000\mu\text{s}$ . Our advantages are even more distinctive considering the fact that GDA ensures serializability, while JanusGraph uses its default configuration with a more relaxed eventual consistency. Neo4j is slower than both GDA and JanusGraph; it however shows similar trends in the differences between particular operation types (e.g., read operations are on average faster than updates). While most Neo4j operations finish below 20ms, it does entail relatively many outliers.

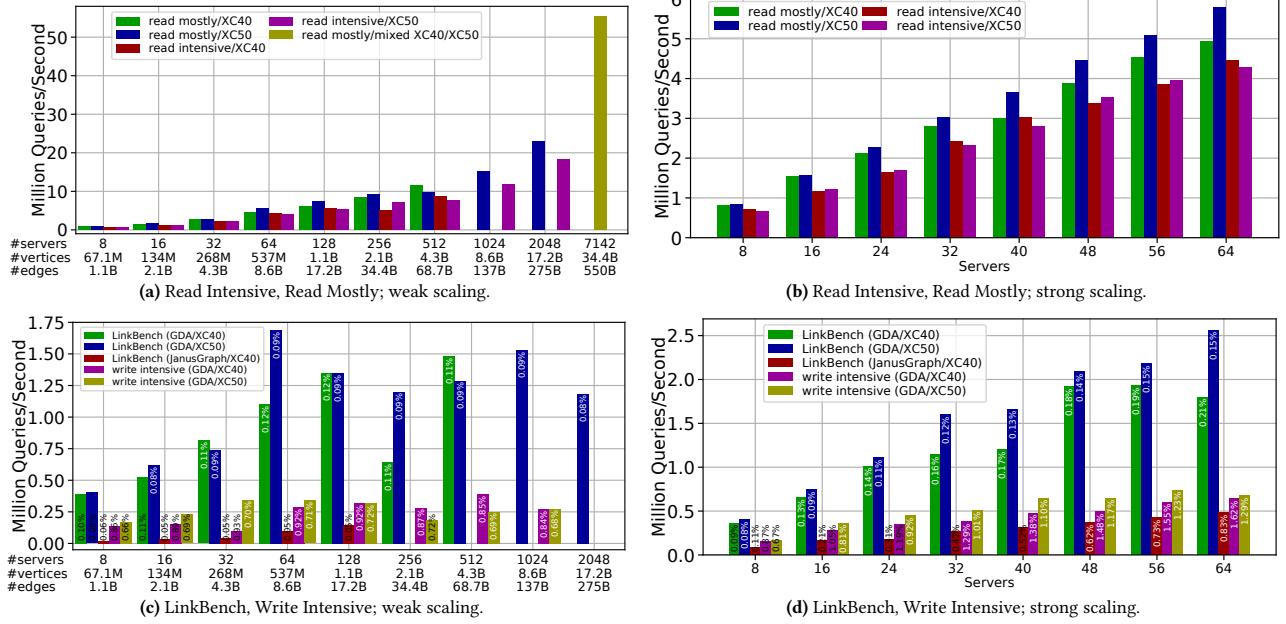
**Summary of GDA’s Advantages** GDA is faster than comparison targets due to its fundamental reliance on one-sided RDMA.

## 6.5 Analysis of OLAP and OLSP Workloads

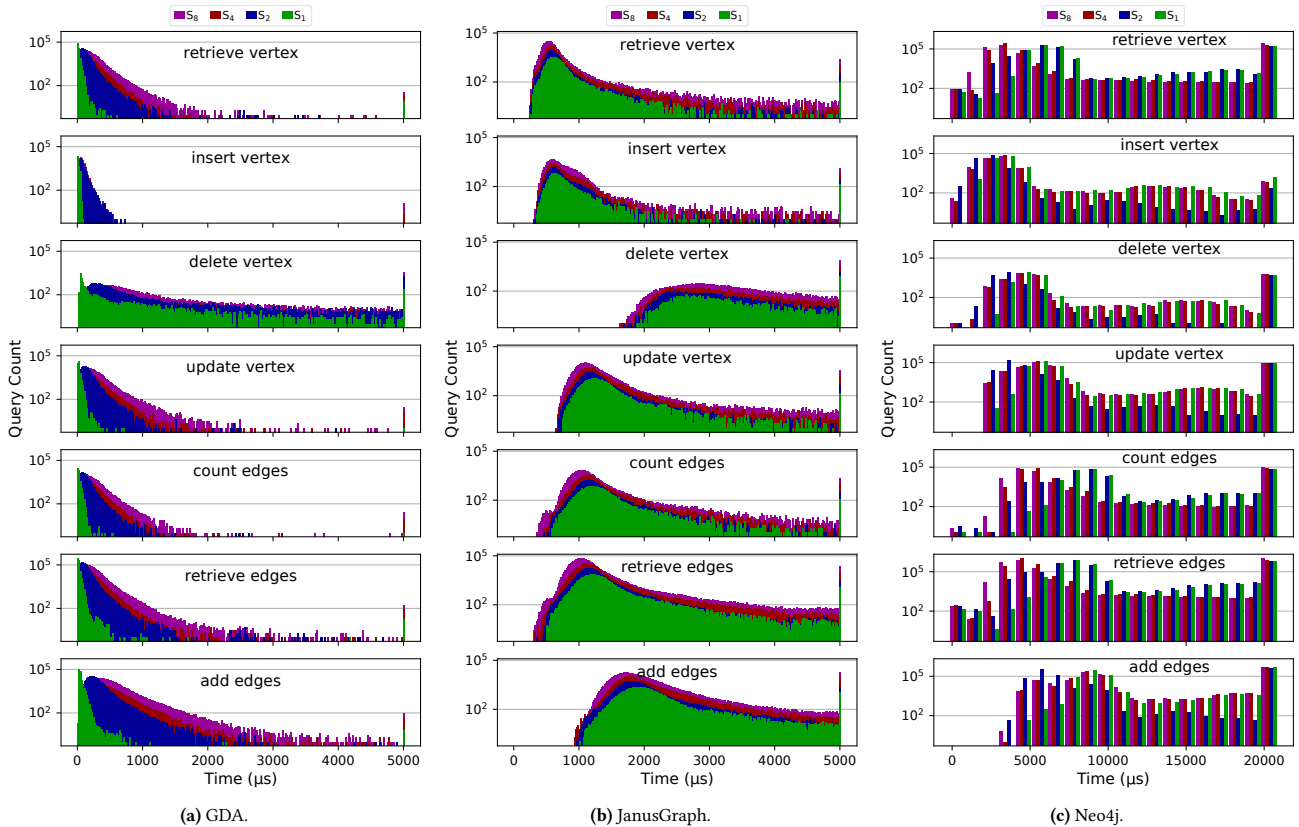
We illustrate the OLAP and OLSP results in Figure 6. We consider BFS, PageRank (PR), Community Detection using Label Propagation (CDLP), Weakly Connected Components (WCC), Local Cluster Coefficient (LCC), Business Intelligence 2 query from LDDB SNB (BI2) [97], and Graph Neural Networks (GNN; training of the graph convolution model [59]). The results follow advantageous performance patterns – for most problems (BFS, k-hop, GNN) adding more compute resources combined with increasing the dataset size only results in mild runtime increases (in weak scaling) or runtime drops (in strong scaling). WCC, CDLP, and PR are characterized by overall sharper slopes of increasing running times for weak scaling; we conjecture this is because these problems cumulatively involve more communication due to their memory access patterns and runtime complexities (e.g., LCC has the complexity of  $O(n + m^{3/2})$  compared to  $O(m + n)$  for BFS).

GDA also outperforms other graph databases in OLAP/OLSP by large margins. Here, to also compare to more competitive targets, we consider the Graph500 implementation of BFS [76]. It is a highly

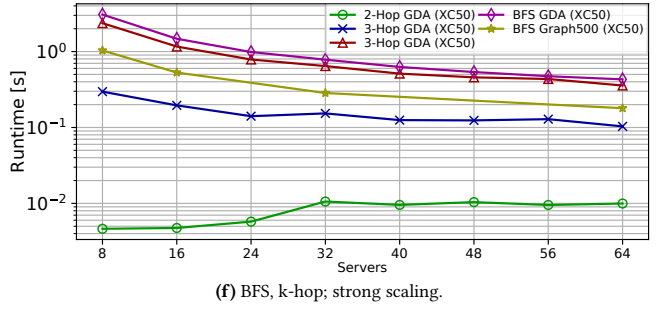
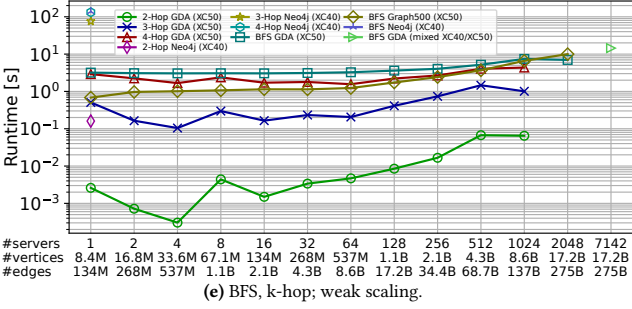
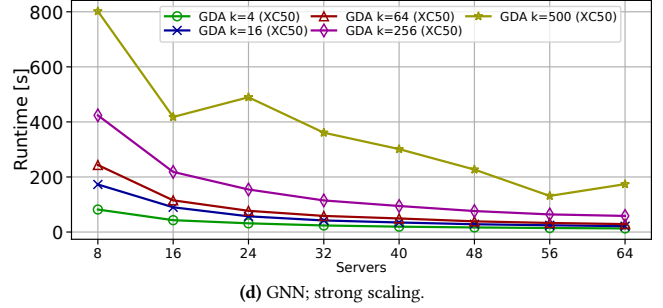
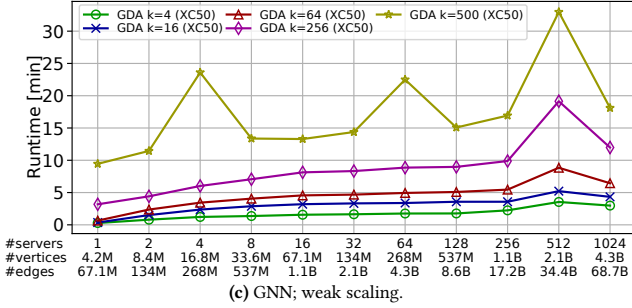
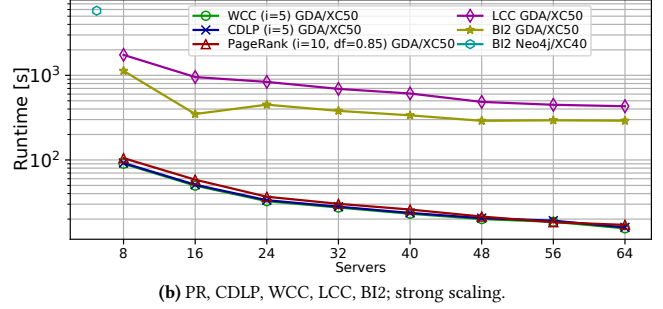
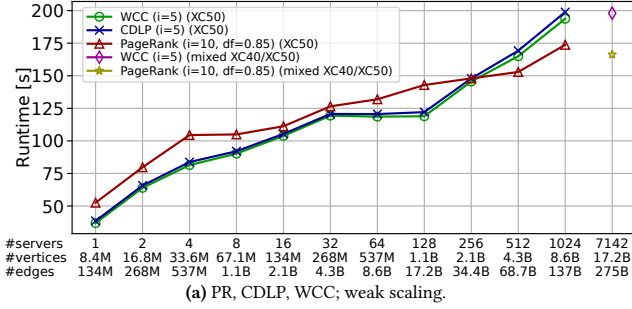




**Figure 4:** Analysis of OLTP workloads. **XC40, XC50:** two types of servers considered (cf. Section 6.1). **Weak scaling:** scaling dataset sizes together with #servers, **strong scaling:** scaling #servers for a fixed dataset (a Kronecker graph of scale 26, i.e., 67.1M vertices and 1.1B edges; the results follow the same performance patterns for other datasets). **Missing bars** of our baselines indicate limited compute budget; missing baselines of comparison targets indicate inability to scale to a given configuration. **Percentages:** the fractions of failed transactions (no percentage indicates no, or negligibly few, failed transactions).



**Figure 5:** Details (histograms) on the latency of individual operations of the OLTP LinkBench workload for 1, 2, 4 and 8 servers. We aggregate query latencies outside the range and plot their combined number at the upper bound. **Note the different cutoff of the X axis for Neo4j (20 ms) and GDA as well as JanusGraph (5 ms).** Because Neo4j only supports a coarser granularity of milliseconds (versus μs for GDA and JanusGraph) for query time measurements, we use a different cutoff and style for plotting. S<sub>x</sub> indicates the data from a specific number of servers.



**Figure 6:** Analysis of OLAP and OLSP workloads. **PR:** PageRank, **CDLP:** Community Detection using Label Propagation, **WCC:** Weakly Connected Components, **LCC:** Local Cluster Coefficient, **BI2:** Business Intelligence 2 query from LDBC SNB, **GNN:** Graph Neural Networks (training of the graph convolution model), **weak scaling:** scaling dataset sizes together with #servers, **strong scaling:** scaling #servers for a fixed dataset (a Kronecker graph of scale 26, i.e., 67.1M vertices and 1.1B edges; the results follow the same performance patterns for other datasets). **Missing data points** of our baselines indicate limited compute budget; missing baselines of comparison targets indicate inability to scale to a given configuration; isolated GDA data points not connected with lines to the rest of the data series indicate extreme-scale runs.

tuned BFS code that has been used for many years to assess high-performance clusters in their abilities to process graph traversals. Graph500 uses graphs with no labels or properties, and it does not use graph transactions. Importantly, GDA is at most 2–4× slower than Graph500, and sometimes it is comparable or even faster (e.g., see 2,048 servers for weak scaling). Hence, GDA is able to deliver high performance graph analytics of even largest scales considered.

**Summary of GDA’s Advantages** Using MPI collectives gives GDA significant benefits for OLAP/OLSP queries. As collectives offer clear semantics, they further boost performance by eliminating boilerplate code.

### 6.6 Varying Labels, Properties, & Edge Factors

In addition to scaling graph sizes (#vertices and #edges), we also analyze GDA’s performance for graphs with different amounts of labels and properties. Intuitively, graphs with very few of these have little rich data attached to vertices and edges. Thus, workloads are mostly dominated by irregular distributed memory single-block reads and writes. With more labels and properties, data accesses are still irregular (due to the nature of graph workloads), but reads and

writes may access many blocks. GDA’s advantages are preserved in all these cases, thanks to harnessing the underlying RDMA.

We use the default value of the edge factor  $e = 16$ , which results in Kronecker graphs close to many real-world datasets in terms of their degree distribution and sparsity. We also tried other values of  $e$ , they also come with similar GDA’s advantages.

### 6.7 Analysis of Real-World Graphs

We also consider large real-world graphs, (which includes Web Data Commons and other largest publicly available real-world datasets) selected from the KONECT [62] and WebGraph [22] repositories. The performance patterns and GDA’s advantages are similar to those obtained for Kronecker graphs. This is because both the considered real-world and Kronecker graphs have similar sparsities as well as heavy-tail degree distributions that have been identified as key factors that determine performance patterns. For example, we were able to process an OLAP BFS query on the Web Data Commons dataset with  $\approx 3.56$  billion vertices and  $\approx 128$  billion edges in  $\approx 15$ s using 1,024 XC50 servers.

## 6.8 Extreme Scales & Comparison to Others

Our evaluation comes with the largest experiments described in the literature in terms of #servers, #cores, and #edges. These largest runs are pictured in Figure 4a for OLTP (RM), and in Figures 6a and 6e for OLAP (WCC, PR, BFS). We were only able to run a few such experiments due to the fact that it required using the full scale of the Piz Daint supercomputer. The results illustrate that even at such workloads, GDA still offers high scalability. For example, moving from a graph with 275B edges to 550B edges increases the OLTP throughput by  $\approx 3\times$  while #servers increase by 3.49 $\times$ .

One recent study with large-scale executions is from the commercial ByteGraph system [65]. However, it does not specify the details of the used graph, and it partially uses disks. Second, while a recent study of the TigerGraph commercial system comes with a graph of a similar size to us (539.6B edges) [100], their servers have significantly more memory (each has  $\approx 1\text{TB}$  vs. 64 GB in our setting). As the network is the main bottleneck in large-scale communication, we expect that GDA would also scale well with such fat-memory servers, and thus it could be able to scalably process even larger graphs than the ones we tried.

Finally, note that our runs required using both XC40 and XC50 servers simultaneously to use full Piz Daint's scale. As XC40 and XC50 come with different CPUs and core counts, this may cause load imbalance. Thus, we conjecture that when using GDA in production data centers with uniform servers, its performance and scalability could be even better than described in this work.

## 7 RELATED WORK

GDBs have been researched in both academia and industry [6, 7, 33, 41, 47, 60, 61], in terms of query languages [3, 4, 23], database management [23, 54, 73, 84, 85], execution in novel environments such as the serverless setting [69, 101], and others [34]. Many graph databases exist [2, 9–11, 20, 26, 27, 29, 32, 35, 36, 39, 40, 56, 57, 67, 70–72, 77–81, 86, 88, 89, 95, 98–100, 104, 107, 108]. In this context, GDI offers standardized building blocks for GDBs to foster portability and programmability across different architectures.

Many **workload specifications and benchmarks for GDBs** exist, covering OLTP interactive queries (SNB [37], LinkBench [12], and BG [13]), OLAP workloads (Graphalytics [53]), or business intelligence queries (BI [96, 97]). One can express these workloads using portable and programmable GDI building blocks. Note that global analytics workloads are the focus of Pregel-like systems [31, 44, 68]. These systems are mostly incomparable to GDBs because they do not support graph updates or LPG datasets.

**Resource Description Framework (RDF)** [63] is a standard to encode knowledge in ontological models and in RDF stores using triples [48, 74, 83]. We focus on graph databases built on top of LPG, and thus RDF designs are outside the scope of this work.

## 8 CONCLUSION

Graph databases (GDBs) are of central importance in academia and industry, and they drive innovation in many domains ranging from computational chemistry to engineering. However, with extreme-scale graphs on the horizon, they face several challenges, including high performance, scalability, programmability, and portability.

In this work, we provide the first systematic approach to address these challenges. First, we design the Graph Database Interface


(GDI): an MPI-inspired specification of performance-critical building blocks for the transactional and storage layer of a GDB. By incorporating the established MPI principles into the GDB domain, we enable designing GDBs that are portable, have well-defined behavior, and seamlessly incorporate workloads as diverse as OLTP, OLAP, and OLSP. Moreover, while in the current GDI release we focus on Labeled Property Graphs, GDI can be straightforwardly extended to cover other data models such as RDF or Knowledge Graphs. This would further illustrate the applicability of HPC-based design principles even well beyond traditional graph databases.

To illustrate the potential of GDI in practice, we use it to build GDI-RMA, a graph database for distributed-memory RDMA architectures. To the best of our knowledge, GDI-RMA is the first GDB that harnesses many powerful HPC mechanisms, including collective communication, offloaded RDMA, non-blocking communication, and network-accelerated atomics. We crystallize the most important design decisions into generic comprehensive insights that can be reused for developing other high-performance GDBs.

In evaluation, we achieve unprecedented performance and scalability for a plethora of workloads, including diverse small transactional queries as well as large graph analytics such as Community Detection or Graph Neural Networks. GDI-RMA outperforms not only other graph databases by orders of magnitude, but its implementation of BFS approaches and even matches in some cases Graph500, a high-performance graph traversal implementation tuned over many years. This is an important result, because the Graph500 kernel only implements the single BFS algorithm for static simple graphs without any rich data, while GDI-RMA is a GDB engine with transactional support for arbitrary graph modifications, the LPG rich data model, and many types of queries.

Finally, we deliver the largest experiments reported in the GDB literature in terms of #servers and #cores, improving upon previous results by orders of magnitude. Our code is publicly available and can be used to propel developing next-generation GDBs.

## ACKNOWLEDGEMENTS

We thank Hussein Harake, Colin McMurtrie, Mark Klein, Angelo Mangili, Marco Induni, Andreas Jocksch, Maria Grazia Giuffreda and the whole CSCS team granting access to the Ault and Daint machines, and for their excellent technical support. We thank Timo Schneider for immense help with infrastructure at SPCL, and PRODYNA AG (Darko Križić, Jens Nixdorf, Christoph Körner) for generous support. We thank Emanuel Peter, Claude Barthels, Jakub Jałowiec, Roman Haag, and Jan Kleine, for help with the project. This project received funding from the European Research Council  (Project PSAP, No. 101002047), and the European High-Performance Computing Joint Undertaking (JU) under grant agreement No. 955513 (MAELSTROM). This project was supported by the ETH Future Computing Laboratory (EFCL), financed by a donation from Huawei Technologies. This project received funding from the European Union's HE research and innovation programme under the grant agreement No. 101070141 (Project GLACIATION). This work was also supported in part by the European Union's Horizon 2020 research and innovation programme under the grant: Sano No. 857533 and the International Research Agendas programme of the Foundation for Polish Science, co-financed by the EU under the European Regional Development Fund.

## References

- [1] Janez Ales. 2020. BASF Enterprise Knowledge Graph Evolution - 55 Billion Entities and Counting. In *Proceedings of NODES (NODES '20)*.
- [2] Amazon. 2018. Amazon Neptune. Available at <https://aws.amazon.com/neptune/>.
- [3] Renzo Angles, Marcelo Arenas, Pablo Barcelo, Peter Boncz, George Fletcher, Claudio Gutierrez, Tobias Lindaaker, Marcus Paradies, Stefan Plantikow, Juan Sequeda, Oskar van Rest, and Hannes Voigt. 2018. G-CORE: A Core for Future Graph Query Languages. In *Proceedings of the International Conference on Management of Data (SIGMOD '18)*. ACM, 1421–1432.
- [4] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. 2017. Foundations of Modern Query Languages for Graph Databases. *ACM Comput. Surv.* 50, 5, Article 68 (Sept. 2017), 40 pages.
- [5] Renzo Angles, Peter Boncz, Josep Larriba-Pey, Irini Fundulaki, Thomas Neumann, Orri Erling, Peter Neubauer, Norbert Martinez-Bazan, Venelin Kotsev, and Ioan Toma. 2014. The Linked Data Benchmark Council: A Graph and RDF Industry Benchmarking Effort. *SIGMOD Rec.* 43, 1 (May 2014), 27–31.
- [6] Renzo Angles and Claudio Gutierrez. 2008. Survey of Graph Database Models. *ACM Comput. Surv.* 40, 1, Article 1 (Feb. 2008), 39 pages.
- [7] Renzo Angles and Claudio Gutierrez. 2018. An Introduction to Graph Data Management. In *Graph Data Management, Fundamental Issues and Recent Developments*, George H. L. Fletcher, Jan Hidders, and Josep Lluís Larriba-Pey (Eds.). Springer, 1–32.
- [8] Apache Software Foundation. 2009. Apache TinkerPop. Available at <http://tinkerpop.apache.org>.
- [9] Apache Software Foundation. 2018. Apache Marmotta. Available at <http://marmotta.apache.org/>.
- [10] Apache Software Foundation. 2021. Apache Jena TBD. Available at <https://jena.apache.org/documentation/tdb/index.html>.
- [11] ArangoDB Inc. 2018. ArangoDB. Available at <https://www.arangodb.com/docs/stable/data-models.html>.
- [12] Timothy G. Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. 2013. LinkBench: A Database Benchmark Based on the Facebook Social Graph. In *Proceedings of the International Conference on Management of Data (SIGMOD '13)*. ACM, 1185–1196.
- [13] Sumita Barahmand and Shahram Ghandeharizadeh. 2013. BG: A Benchmark to Evaluate Interactive Social Networking Actions. In *Proceedings of the 6th Biennial Conference on Innovative Data Systems Research (CIDR '13)*.
- [14] Maciej Besta, Robert Gerstenberger, Nils Blach, Marc Fischer, and Torsten Hoefler. 2023. *GDI: A Graph Database Interface Standard*. Technical Report. Available at <https://github.com/spcl/GDI-RMA>.
- [15] Maciej Besta and Torsten Hoefler. 2015. Accelerating Irregular Computations with Hardware Transactional Memory and Active Messages. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '15)*. ACM, 161–172.
- [16] Maciej Besta and Torsten Hoefler. 2022. Parallel and Distributed Graph Neural Networks: An In-Depth Concurrency Analysis. arXiv:2205.09702
- [17] Maciej Besta, Emanuel Peter, Robert Gerstenberger, Marc Fischer, Michał Podstawski, Claude Barthels, Gustavo Alonso, and Torsten Hoefler. 2023. Demystifying Graph Databases: Analysis and Taxonomy of Data Organization, System Designs, and Graph Queries. *ACM Comput. Surv.* (Jun 2023).
- [18] Maciej Besta, Michał Podstawski, Linus Groner, Edgar Solomonik, and Torsten Hoefler. 2017. To Push or To Pull: On Reducing Communication and Synchronization in Graph Computations. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '17)*. ACM, 93–104.
- [19] Gianfranco Bilardi and Andrea Pietracaprina. 2011. Models of Computation, Theoretical. In *Encyclopedia of Parallel Computing*, David Padua (Ed.). Springer, 1150–1158.
- [20] BlazeGraph. 2018. BlazeGraph DB. Available at <https://www.blazegraph.com/>.
- [21] Guy E. Blelloch and Bruce M. Maggs. 2010. Parallel Algorithms. In *Algorithms and Theory of Computation Handbook: Special Topics and Techniques*. Chapman & Hall/CRC.
- [22] Paolo Boldi and Sebastiano Vigna. 2004. The Webgraph Framework I: Compression Techniques. In *Proceedings of the 13th International Conference on World Wide Web (WWW '04)*. ACM, 595–602.
- [23] Angela Bonifati, George Fletcher, Hannes Voigt, and Nikolay Yakovets. 2018. *Querying Graphs*. Springer.
- [24] Michael M Bronstein, Joan Bruna, Yann LeCun, Arthur Szlam, and Pierre Vandergheynst. 2017. Geometric Deep Learning: Going beyond Euclidean data. *IEEE Signal Processing Magazine* 34, 4 (2017), 18–42.
- [25] Chiranjeev 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 International Conference on Management of Data (SIGMOD '20)*. ACM, 329–344.
- [26] Cambridge Semantics. 2018. AnzoGraph. Available at <https://www.cambridgesemantics.com/anzograph/>.
- [27] Cayley. 2018. CayleyGraph. Available at <https://cayley.io/> and <https://github.com/cayleygraph/cayley>.
- [28] Ernie Chan, Marcel Heimlich, Avi Purkayastha, and Robert van de Geijn. 2007. Collective Communication: Theory, Practice, and Experience: Research Articles. *Concurr. Comput.: Pract. Exper.* 19, 13 (Sept. 2007), 1749–1783.
- [29] 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 Symposium on Cloud Computing (SoCC '19)*. ACM, 87–100.
- [30] Hongzhi Chen, Changji Li, Chenguang Zheng, Chenghuan Huang, Juncheng Fang, James Cheng, and Jian Zhang. 2022. G-Tran: A High Performance Distributed Graph Database with a Decentralized Architecture. *Proc. VLDB Endow.* 15, 11 (July 2022), 2545–2558.
- [31] Rong Chen, Jiaxin Shi, Yanzhe Chen, Binyu Zang, Haibing Guan, and Haibo Chen. 2019. PowerLyra: Differentiated Graph Computation and Partitioning on Skewed Graphs. *ACM Trans. Parallel Comput.* 5, 3, Article 13 (Jan. 2019), 39 pages.
- [32] DataStax Inc. 2018. DSE Graph (DataStax). Available at <https://www.datastax.com/>.
- [33] Ali Davoudian, Liu Chen, and Mengchi Liu. 2018. A Survey on NoSQL Stores. *ACM Comput. Surv.* 51, 2, Article 40 (April 2018), 43 pages.
- [34] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor E. Lee. 2020. Aggregation Support for Modern Graph Analytics in TigerGraph. In *Proceedings of the International Conference on Management of Data (SIGMOD '20)*. ACM, 377–392.
- [35] Dgraph Labs, Inc. 2018. DGraph. Available at <https://dgraph.io/> and <https://dgraph.io/docs/>.
- [36] Ayush Dubey, Greg D. Hill, Robert Escriva, and Emin Gün Sirer. 2016. Weaver: A High-Performance, Transactional Graph Database Based on Refinable Timestamps. *Proc. VLDB Endow.* 9, 11 (July 2016), 852–863.
- [37] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. 2015. The LDBC Social Network Benchmark: Interactive Workload. In *Proceedings of the International Conference on Management of Data (SIGMOD '15)*. ACM, 619–630.
- [38] Greg Faenes, Abdulla Bataneh, Duncan Roweth, Tom Court, Edwin Froese, Bob Alverson, Tim Johnson, Joe Kopnick, Mike Higgins, and James Reinhard. 2012. Cray Cascade: A Scalable HPC System Based on a Dragonfly Network. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12)*. IEEE, Article 103, 9 pages.
- [39] FactNexus. 2018. GraphBase. Available at <https://graphbase.ai/>.
- [40] Franz Inc. 2018. AllegroGraph. Available at <https://allegrograph.com/products/allegrograph/>.
- [41] Santhosh Kumar Gajendran. 2012. A Survey on NoSQL Databases.
- [42] Robert Gerstenberger, Maciej Besta, and Torsten Hoefler. 2013. Enabling Highly-Scalable Remote Memory Access Programming with MPI-3 One Sided. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '13)*. ACM, Article 53, 12 pages.
- [43] Lukas Gianinazzi, Maximilian Fries, Nikoli Dryden, Tal Ben-Nun, Maciej Besta, and Torsten Hoefler. 2021. Learning Combinatorial Node Labeling Algorithms. arXiv:2106.03594
- [44] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI '12)*. 17–30.
- [45] Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Martin Schuster, Petra Selmer, and Hannes Voigt. 2019. Updating Graph Databases with Cypher. *Proc. VLDB Endow.* 12, 12 (Aug. 2019), 2242–2254.
- [46] William L. Hamilton, Rex Ying, and Jure Leskovec. 2017. Representation Learning on Graphs: Methods and Applications. *Bulletin of the Technical Committee on Data Engineering* 40, 3 (Sept. 2017), 52–74.
- [47] Jing Han, E Haihong, Guan Le, and Jian Du. 2011. Survey on NoSQL database. In *Proceedings of the 6th International Conference on Pervasive Computing and Applications (ICPCA '11)*. IEEE, 363–366.
- [48] Steve Harris, Nick Lamb, and Nigel Shadbolt. 2009. 4store: The Design and Implementation of a Clustered RDF Store. In *Proceedings of the 5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS '09)*.
- [49] Maurice Herlihy and Nir Shavit. 2008. *The Art of Multiprocessor Programming*. Morgan Kaufmann.
- [50] Torsten Hoefler and Roberto Belli. 2015. Scientific Benchmarking of Parallel Computing Systems: Twelve Ways to Tell the Masses When Reporting Performance Results. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*. ACM, Article 73, 12 pages.
- [51] Torsten Hoefler and Dmitry Moor. 2014. Energy, Memory, and Runtime Tradeoffs for Implementing Collective Communication Operations. *Supercomputing Frontiers and Innovations* 1, 2 (Sept. 2014), 58–75.

- [52] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. 2008. Accurately measuring collective operations at massive scale. In *Proceedings of the International Symposium on Parallel and Distributed Processing (IPDPS '08)*. IEEE, 1–8.
- [53] Alexandru Iosup, Tim Hegeman, Wing Lung Ngai, Stijn Heldens, Arnau Prat-Pérez, Thomas Manhardt, Hassan Chafio, Mihai Capotă, Narayanan Sundaram, Michael Anderson, Ilie Gabriel Tânase, Yinglong Xia, Lifeng Nai, and Peter Boncz. 2016. LDBC Graphalytics: A Benchmark for Large-Scale Graph Analysis on Parallel and Distributed Platforms. *Proc. VLDB Endow.* 9, 13 (Sept. 2016), 1317–1328.
- [54] Martin Jungmanns, André Petermann, Martin Neumann, and Erhard Rahm. 2017. Management and Analysis of Big Graph Data: Current Systems and Open Challenges. In *Handbook of Big Data Technologies*, Albert Y. Zomaya and Sherif Sakr (Eds.). Springer, 457–505.
- [55] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2016. Design Guidelines for High Performance RDMA Systems. In *Proceedings of the USENIX Annual Technical Conference (ATC '16)*. 437–450.
- [56] U. Kang, Hanghang Tong, Jimeng Sun, Ching-Yung Lin, and Christos Faloutsos. 2012. Gbase: An Efficient Analysis Platform for Large Graphs. *The VLDB Journal* 21, 5 (Oct. 2012), 637–650.
- [57] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedbhi, Jeremy Chen, and Semih Salihoglu. 2017. Graphflow: An Active Graph Database. In *Proceedings of the International Conference on Management of Data (SIGMOD '17)*. ACM, 1695–1698.
- [58] John Kim, Wiliam J. Dally, Steve Scott, and Dennis Abts. 2008. Technology-Driven, Highly-Scalable Dragonfly Topology. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA '08)*. IEEE, 77–88.
- [59] Thomas N Kipf and Max Welling. 2016. Semi-Supervised Classification with Graph Convolutional Networks. arXiv:1609.02907
- [60] Vijay Kumar and Anjan Babu. 2015. Domain Suitable Graph Database Selection: A Preliminary Report. In *Proceedings on the 3rd International Conference on Advances in Engineering Sciences & Applied Mathematics (ICAESAM '15)*. 26–29.
- [61] Rohit kumar Kaliyar. 2015. Graph databases: A survey. In *Proceedings of the International Conference on Computing, Communication & Automation (ICCCA '15)*. IEEE, 785–790.
- [62] Jérôme Kunegis. 2013. KONECT: The Koblenz Network Collection. In *Proceedings of the 22nd International Conference on World Wide Web (WWW '13 Companion)*. ACM, 1343–1350.
- [63] Ora Lassila and Ralph R. Swick. 1998. *Resource Description Framework (RDF) Model and Syntax Specification*.
- [64] Jure Leskovec, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutsos, and Zoubin Ghahramani. 2010. Kronecker Graphs: An Approach to Modeling Networks. *Journal of Machine Learning Research* 11, 33 (Feb. 2010), 985–1042.
- [65] Changji Li, Hongzhi Chen, Shuai Zhang, Yingqian Hu, Chao Chen, Zhenjie Zhang, Meng Li, Xiangchen Li, Dongqing Han, Xiaohui Chen, Xudong Wang, Huiming Zhu, Xuwei Fu, Tingwei Wu, Hongfei Tan, Hengtian Ding, Mengjin Liu, Kangcheng Wang, Ting Ye, Lei Li, Xin Li, Yu Wang, Chenguang Zheng, Hao Yang, and James Cheng. 2022. ByteGraph: A High-Performance Distributed Graph Database in ByteDance. *Proc. VLDB Endow.* 15, 12 (Aug. 2022), 3306–3318.
- [66] Heng Lin, Xiaowei Zhu, Bowen Yu, Xiongchao Tang, Wei Xue, Wenguang Chen, Lufei Zhang, Torsten Hoefler, Xiaosong Ma, Xin Liu, Weimin Zheng, and Jingfang Xu. 2018. ShenTu: Processing Multi-Trillion Edge Graphs on Millions of Cores in Seconds. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '18)*. IEEE, Article 56, 11 pages.
- [67] Linux Foundation. 2018. JanusGraph. Available at <http://janusgraph.org/>.
- [68] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A System for Large-Scale Graph Processing. In *Proceedings of the International Conference on Management of Data (SIGMOD '10)*. ACM, 135–146.
- [69] Zhitao Mao, Ruoyu Wang, Haoran Li, Yixin Huang, Qiang Zhang, Xiaoping Liao, and Hongwu Ma. 2022. ERMer: a serverless platform for navigating, analyzing, and visualizing Escherichia coli regulatory landscape through graph database. *Nucleic Acids Research* 50, W1 (April 2022), W298–W304.
- [70] Norbert Martínez-Bazan, M. Ángel Águila Lorente, Víctor Muntés-Mulero, David Dominguez-Sal, Sergio Gómez-Villamor, and Josep-L. Larriba-Pey. 2012. Efficient Graph Management Based On Bitmap Indices. In *Proceedings of the 16th International Database Engineering & Applications Symposium (IDEAS '12)*. ACM, 110–119.
- [71] Memgraph Ltd. 2018. Memgraph. Available at <https://memgraph.com/>.
- [72] Microsoft. 2018. Azure Cosmos DB. Available at <https://azure.microsoft.com/en-us/services/cosmos-db/>.
- [73] Justin J. Miller. 2013. Graph Database Applications and Concepts with Neo4j. In *Proceedings of the Southern Association for Information Systems Conference (SAIS '13)*.
- [74] Gianfranco E. Modoni, Marco Sacco, and Walter Terkaj. 2014. A survey of RDF store solutions. In *Proceedings of the International Conference on Engineering, Technology and Innovation (ICE '14)*. IEEE, 1–7.
- [75] MPI Forum. 2012. *MPI: A Message-Passing Interface Standard, version 3*.
- [76] Richard C. Murphy, Kyle B. Wheeler, Brian W. Barrett, and James A. Cug. 2010. Introducing the Graph 500. In *Proceedings of the Cray User Group (CUG '10)*. 45–74.
- [77] Networked Planet Limited. 2018. BrightstarDB. Available at <http://brightstardb.com/>.
- [78] Objectivity Inc. 2018. InfiniteGraph. Available at <https://www.objectivity.com/products/infinitegraph/>.
- [79] Ontotext. 2018. GraphDB. Available at <https://www.ontotext.com/products/graphdb/>.
- [80] OpenLink. 2018. Virtuoso. Available at <https://virtuoso.openlinksw.com/>.
- [81] Oracle. 2018. Oracle Spatial and Graph. Available at <https://www.oracle.com/database/technologies/spatialandgraph.html>.
- [82] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank Citation Ranking: Bringing Order to the Web*. Technical Report. Stanford InfoLab.
- [83] Nikolaos Papailiou, Ioannis Konstantinou, Dimitrios Tsoumakos, and Nectarios Koziris. 2012. H2RDF: Adaptive Query Processing on RDF Data in the Cloud.. In *Proceedings of the 21st International Conference on World Wide Web (WWW '12 Companion)*. ACM, 397–400.
- [84] N.S. Patil, P Kiran, N.P. Kavya, and K.M. Naresh Patel. 2018. A Survey on Graph Database Management Techniques for Huge Unstructured Data. *International Journal of Electrical and Computer Engineering* 81, 2 (2018), 1140–1149.
- [85] Jaroslav Pokorný. 2015. Graph Databases: Their Power and Limitations. In *Proceedings of the IFIP International Conference on Computer Information Systems and Industrial Management (CISIM '15)*, Khalid Saeed and Wladyslaw Homenda (Eds.). Lecture Notes in Computer Science, Vol. 9339. Springer, 58–69.
- [86] Profium. 2018. Profium Sense. Available at <https://www.profium.com/en/products/graph-database/>.
- [87] Zhengping Qian, Chenqiang Min, Longbin Lai, Yong Fang, Gaofeng Li, Youyang Yao, Bingqing Lyu, Xiaoli Zhou, Zhimin Chen, and Jingren Zhou. 2021. GALA: A System for Interactive Analysis on Distributed Graphs Using a High-Level Language. In *Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI '21)*. 321–335.
- [88] Redis Labs. 2018. RedisGraph. Available at <https://redis.io/docs/stack/graph/>.
- [89] Christopher D. Rickett, Utz-Uwe Haus, James Maltby, and Kristyn J. Maschhoff. 2018. Loading and Querying a Trillion RDF triples with Cray Graph Engine on the Cray XC. In *Proceedings of the Cray User Group (CUG '18)*.
- [90] Ian Robinson, Jim Webber, and Emil Eifrem. 2015. Graph Database Internals. In *Graph Databases* (2nd ed.). O'Reilly, Chapter 7, 149–170.
- [91] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2008. The Graph Neural Network Model. *IEEE Transactions on Neural Networks* 20, 1 (2008), 61–80.
- [92] Hermann Schweizer, Maciej Besta, and Torsten Hoefler. 2015. Evaluating the Cost of Atomic Operations on Modern Architectures. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT '15)*. IEEE, 445–456.
- [93] Jiaxin Shi, Youyang Yao, Rong Chen, Haibo Chen, and Feifei Li. 2016. Fast and Concurrent RDF Queries with RDMA-Based Distributed Graph Exploration. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*. 317–332.
- [94] Anna Kornfeld Simpson, Adriana Szekeres, Jacob Nelson, and Irene Zhang. 2020. Securing RDMA for High-Performance Datacenter Storage Systems. In *Proceedings of the 12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '20)*.
- [95] Stardog Union. 2018. Stardog. Available at <https://www.stardog.com/>.
- [96] Gábor Szárnyas, Arnau Prat-Pérez, Alex Averbuch, József Marton, Marcus Paradies, Moritz Kaufmann, Orri Erling, Peter Boncz, Vlad Haprian, and János Benjamin Antal. 2018. An Early Look at the LDBC Social Network Benchmark's Business Intelligence Workload. In *Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA) (GRADES-NDA '18)*. Article 9, 11 pages.
- [97] Gábor Szárnyas, Jack Waudby, Benjamin A. Steer, Dávid Szakállas, Altan Birler, Mingxi Wu, Yuchen Zhang, and Peter Boncz. 2022. The LDBC Social Network Benchmark: Business Intelligence Workload. *Proc. VLDB Endow.* 16, 4 (Dec. 2022), 877–890.
- [98] Claudio Tesoriero. 2013. *Getting Started with OrientDB*. Packt Publishing.
- [99] TigerGraph Inc. 2018. TigerGraph. Available at <https://www.tigergraph.com/>.
- [100] TigerGraph Inc. 2022. *Using the Linked Data Benchmark Council Social Network Benchmark Methodology to Evaluate TigerGraph at 36 Terabytes*. White Paper.
- [101] Lucian Toader, Alexandru Uta, Ahmed MUSAafir, and Alexandru Iosup. 2019. Graphless: Toward Serverless Graph Processing. In *Proceedings of the 18th International Symposium on Parallel and Distributed Computing (ISPD '19)*. IEEE, 66–73.

- [102] Yandong Wang, Li Zhang, Jian Tan, Min Li, Yuqing Gao, Xavier Guerin, Xiaoqiao Meng, and Shicong Meng. 2015. HydraDB: A Resilient RDMA-Driven Key-Value Middleware for in-Memory Cluster Computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*. ACM, Article 22, 11 pages.
- [103] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. 2021. A Comprehensive Survey on Graph Neural Networks. *IEEE Transactions on Neural Networks and Learning Systems* 32, 1 (2021), 4–24.
- [104] Pingpeng Yuan, Pu Liu, Buwen Wu, Hai Jin, Wenya Zhang, and Ling Liu. 2013. TripleBit: A Fast and Compact System for Large Scale RDF Data. *Proc. VLDB Endow.* 6, 7 (May 2013), 517–528.
- [105] Ziwei Zhang, Peng Cui, and Wenwu Zhu. 2022. Deep Learning on Graphs: A Survey. *IEEE Transactions on Knowledge and Data Engineering* 34, 1 (2022), 249–270.
- [106] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. 2020. Graph neural networks: A review of methods and applications. *AI Open* 1 (2020), 57–81.
- [107] Xiaowei Zhu, Guanyu Feng, Marco Serafini, Xiaosong Ma, Jiping Yu, Lei Xie, Ashraf Aboulnaga, and Wenguang Chen. 2020. LiveGraph: A Transactional Graph Storage System with Purely Sequential Adjacency List Scans. *Proc. VLDB Endow.* 13, 7 (March 2020), 1020–1034.
- [108] Lei Zou, M. Tamer Özsu, Lei Chen, Xuchuan Shen, Ruizhe Huang, and Dongyan Zhao. 2014. GStore: A Graph-Based SPARQL Query Engine. *The VLDB Journal* 23, 4 (Aug. 2014), 565–590.

# Appendix: Artifact Description/Artifact Evaluation

## ARTIFACT DOI

10.5281/zenodo.8081855

## ARTIFACT IDENTIFICATION

Graph databases (GDBs) are crucial in academic and industry applications. The key challenges in developing GDBs are achieving high performance, scalability, programmability, and portability. To tackle these challenges, we harness established practices from the HPC landscape to build a system that outperforms all past GDBs presented in the literature by orders of magnitude. To achieve this, we crystallize the fundamental performance-critical building blocks of GDBs into a portable and programmable specification called the Graph Database Interface (GDI), inspired by the best practices of MPI. Considering a software stack of (a) storage backend, (b) storage and transaction engine, (c) middle layer (query planing, workload distribution and result aggregation) and (d) external database interface (graph queries), GDI provides a generic and programmable API for the storage and transaction engine layer. It also possible to use GDI to directly implement specific graph queries by the user.

We offer a high-performance implementation of GDI for distributed-memory systems supporting RDMA-enabled interconnects, called GDI-RMA. We implement the GDI API as a C library based on MPI one-sided communication. The library will be published as free software.

We support nearly any function in our implementation with a theoretical performance analysis that is independent of the underlying hardware and thus offers portable performance insights.

We illustrate how to use GDI to program many graph database workloads, covering OLTP, OLAP, and OLSP, which form the bulk of the associated workloads.

We also develop an in-memory distributed generator that can rapidly create graphs of arbitrary size as well as vertex and edge labels and properties. The edge generation is based on the popular Kronecker graph generation from the established Graph500 benchmark<sup>1</sup>. Each process generates the edges of a portion of the graph, which are redistributed afterwards based on the location of the incident vertices of each edge. Vertices are assigned labels based on a configurable percentage. These labels also act as a kind of vertex type and the property type of the properties assigned to a vertex are based on that type, whereas the property value is randomly generated. The label of the edges is chosen based on the labels of the incident vertices.

The evaluation of GDI-RMA significantly surpasses in scale previous GDB analyses in the literature in the counts of compute nodes, counts of cores, and in the size of a single analytic workload.

Detailed description of the computational artifacts and how they contribute to the reproducibility is described in the following "Reproducibility of Experiments".

## REPRODUCIBILITY OF EXPERIMENTS

### Experimental Workflow

The GDI-RMA implementation is provided as a C library, based on MPI, so that is possible to link an application based on GDI against this implementation.

We further provide a benchmark infrastructure. Edges can either be loaded from files in the edgelist format or generated using the Kronecker graph generator from the Graph500 benchmark. The edges can be directed or undirected. In the next step, the vertices and edges are labeled and properties are added to the vertices. It is possible to customize the label and property assignment with the help of user-specified code. Afterwards the resulting labeled property graph can be benchmarked with exemplary implementations of various workloads. It is possible to just generate the labeled property graph and write the graph to CSV files, which can be used for other baseline systems.

The **OLTP workloads** (a detailed description of these workloads can be found in the submission on page 7 table 3) are as follows:

- "read mostly"
- "read intensive"
- "write intensive"
- "LinkBench" (i.e., the established LinkBench benchmark workload).

Since these workloads update the graph, we restart the benchmarking each time with a fresh graph. It possible to measure the latency of the individual queries or the total throughput.

The **OLAP and OLSP** workloads are as follows:

- Breadth-First Search (BFS)
- Business Intelligence Query 2 (BI2)
- Community Detection using Label Propagation (CDLP)
- Graph Neuronal Networks (GNN)
- k-hop
- Local Cluster Coefficient (LCC)
- PageRank (PR)
- Weakly Connected Components (WCC)

BFS, CDLP, LCC, PR and WCC were chosen from the LDBC Graphalytics benchmark and the Business Intelligence query 2 from the LDBC Social Network Benchmark (SNB). We augmented these OLAP and OLSP workloads with a state of the art GNN workload.

A different binary is created for each workload. The graph parameters (number of vertices, edge factor, directed/undirected), certain GDI-RMA specific parameters (amount of reserved main memory, block size) as well as various parameters specific to certain workloads can be set by command line parameters.

After an extensive investigation and configuration effort, we were able to successfully configure and use Neo4j and JanusGraph. These two systems are two highest-ranking core graph databases (i.e., systems with the database model "Graph") in the established DB-Engines Ranking<sup>2</sup> (at the moment of the submission).

<sup>1</sup>[https://graph500.org/?page\\_id=47](https://graph500.org/?page_id=47)

<sup>2</sup><https://db-engines.com/en/ranking/graph+dbms>

## Estimation of Execution Time

We estimate that it takes two weeks to run all necessary experiments presented in the evaluation section.

## Expected Results and Relationship to the Results in the Submission

The expected results from the artifact should exactly match those provided in the submission, assuming using the same SW and HW configuration. Thus, we now describe the detailed configuration needed for generating these results. Using different HW would provide results with the same performance trends (i.e., with similar relative differences between respective baselines). We first provide the parametrization, referring directly to the results in the submission (the results obtained from the artifact match these results).

When using the Kronecker graph generator, the number of vertices is provided as a scale parameter, which is the logarithm base two of that number. The number of edges is determined by an edge factor, which corresponds to the average vertex degree. So the number of edges in the Kronecker graph is the number of vertices multiplied by the edge factor.

For the latency and throughput figures of the OLTP, we executed 10100 queries, and used the first 100 queries as warm up. We usually executed the OLAP and OLSP queries 10 times.

Figure 4a and 4c: Weak scaling OLTP (read mostly/read intensive/LinkBench/write intensive) throughput for Kronecker graphs with an edge factor of 16, which is the default value of the Graph500 benchmark. We benchmarked XC40 and XC50 compute nodes.

- 8 compute nodes/scale = 26
- 16 compute nodes/scale = 27
- 32 compute nodes/scale = 28
- 64 compute nodes/scale = 29
- 128 compute nodes/scale = 30
- 256 compute nodes/scale = 31
- 512 compute nodes/scale = 32
- 1024 compute nodes/scale = 33
- 2048 compute nodes/scale = 34
- 7142 compute nodes/scale = 35

Figures 4b and 4b: Strong scaling OLTP (read mostly/read intensive/LinkBench/write intensive) throughput for a Kronecker graph of scale 26 and an edge factor of 16 with 8, 16, 24, 32, 40, 48, 56 and 64 compute nodes. We benchmarked XC40 and XC50 compute nodes.

Figure 5: Latency of the LinkBench OLTP workload for a Kronecker graph of scale 23 and an edge factor of 16 with 1, 2, 4 and 8 XC40 compute nodes.

Figure 6: workload parameters

**CDLP** : 5 iterations

**GNN** : 5 neural GNN layers and feature vector sizes of 4, 16, 64, 256 and 500

**k-hop** : 2, 3 and 4 hops

**PR** : 10 iterations and a damping factor of 0.85, the default value of the command line parameter

**WCC** : 5 iterations

Figure 6a, 6c, 6e: Weak scaling query runtime of OLAP and OLSP workloads (BFS, CDLP, GNN, k-hop, PR, WCC) for Kronecker

graphs with an edge factor of 16 for XC50 compute nodes. The GNN workload experiments reduced the scale by one to account for the additional storage space of the feature vectors.

- 1 compute nodes/scale = 23
- 2 compute nodes/scale = 24
- 4 compute nodes/scale = 25
- 8 compute nodes/scale = 26
- 16 compute nodes/scale = 27
- 32 compute nodes/scale = 28
- 64 compute nodes/scale = 29
- 128 compute nodes/scale = 30
- 256 compute nodes/scale = 31
- 512 compute nodes/scale = 32
- 1024 compute nodes/scale = 33

BFS results for 2048 (XC50) and 7142 (mixed XC40/XC50) compute nodes with a scale of 34 are additionally provided as well as XC40 results for 8 to 1024 compute nodes. WCC and PageRank results are also provided for 7142 (mixed XC40/XC50) compute nodes with a scale of 34.

Figure 6b, 6d, 6f: Strong scaling query runtime of OLAP and OLSP workloads (BFS, BI2, CDLP, GNN, k-hop, LCC, PR, WCC) of scale 26 and an edge factor of 16 with 8, 16, 24, 32, 40, 48, 56 and 64 XC50 compute nodes. LCC was executed with a Kronecker graph of scale 23 and an edge factor of 16, whereas GNN was executed with a Kronecker graph of scale 25 and an edge factor of 16.

6.7: We executed the BFS query on the Web Data Commons dataset (2012)<sup>3</sup> with  $\approx 3.56$  billion vertices and  $\approx 128$  billion edges in  $\approx 15$ s using 1024 XC50 compute nodes.

The results for the above figures, and the figures themselves, can be generated using the described and provided reproducibility-focused infrastructure described in the Artifact Installation & Deployment Process paragraphs. We ensure that all the scripts and the general workflow are easy to follow and use.

## ARTIFACT DEPENDENCIES REQUIREMENTS

GDI-RMA doesn't required any specific hardware. For the experiments presented in the submission, we used foMPI<sup>4</sup> for one-sided MPI communication, which can only be used on Cray Systems with Gemini and Aries networks.

GDI-RMA was written for Linux-based environments.

GDI-RMA uses MPI to satisfy its communication needs. Otherwise the GDI-RMA library itself has no other dependencies. The benchmark implementation uses LibSciBench<sup>5</sup> for its measurements. Additionally the BFS- and k-hop implementations use output from a modified version of the Graph500 BFS reference implementation as input to match the start vertices of the queries for a fair comparison.

The evaluation of GDI-RMA uses mostly an in-memory Kronecker graph generator based on the Graph500, that generates and distributes the necessary edges on the fly during the initial start up while creating the graph database. Labels and vertex properties are generated randomly based on a user-chosen data scheme. Kronecker graphs were chosen since they emulate realistic real

<sup>3</sup><http://webdatacommons.org/hyperlinkgraph/index.html>

<sup>4</sup>[https://spcl.inf.ethz.ch/Research/Parallel\\_Programming/foMPI/](https://spcl.inf.ethz.ch/Research/Parallel_Programming/foMPI/)

<sup>5</sup><https://spcl.inf.ethz.ch/Research/Performance/LibLSB/>



world graphs with their heavy-tail skewed degree distribution. We developed our own graph generator, since existing one either didn't scale high enough or experienced problems on Piz Daint while generating the necessary data.

Additionally we provide results for one real world dataset: Web Data Commons with  $\approx 3.56$  billion vertices and  $\approx 128$  billion edges, which is publicly available. We choose this dataset to provide evidence of our bulk loading capabilities and to present results on real world dataset, not just synthetic ones.

We use the highly optimized Graph500 BFS reference implementation as a baseline for the comparison with our GDI-RMA BFS workload implementation. Neo4j and JanusGraph provide mainly a baseline for the comparison of the OLTP workloads.

## ARTIFACT INSTALLATION DEPLOYMENT PROCESS

### Installation: foMPI Library (optional)

time: a few minutes

foMPI was used in the evaluation of GDI-RMA on Piz Daint. However foMPI only works with Cray Gemini and Aries networks. This step is optional, you can also use the vendor-supplied MPI library.

```
% cd GDI_artifact
% tar xfz foMPI-0.2.2.tar.gz
% cd foMPI-0.2.2
% edit Makefile.inc
  - default values for CC, FC and CXX should be correct
    for Cray systems
  - add -fcommon to CFLAGS (line 5)
% make libfompi.a
end result: libfompi.a
```

### Installation: GDI-RMA Library

time: a few minutes

```
% cd GDI_artifact
% edit Makefile.inc
  - set CC and CXX to the MPI C- and C++-compilers of
    your system
% cd src
% edit Makefile.inc
  - only necessary if you want to use foMPI: uncomment
    lines 6 to 9
% make
end result: libgdi.a in src
```

### Installation: LibSciBench Library

time: a few minutes

LibSciBench is a library for time measurements, that is used by the benchmark source code. Each rank stores its measurements in a separate file: `lsb.*.r{rank number}`.

```
% cd GDI_artifact
% git clone https://github.com/spcl/liblsb.git
% cd liblsb
% git checkout 91d073d7420b4c4d8d30dab202166f9de7d65a10
% mkdir build
```

```
% module switch PrgEnv-cray PrgEnv-gnu
  - necessary step on Piz Daint
% MPICC=cc MPICXX=CC ./configure --prefix=$(pwd)/build/
  --with-mpi --without-papi
  - set MPICC and MPICXX to the MPI C- and C++-compilers
    of your system
% make
% make install
% module switch PrgEnv-gnu PrgEnv-cray
  - necessary step on Piz Daint
end result: liblsb/build - lib and includes directories
```

### Installation: Benchmarks

time: a few minutes

```
% cd GDI_artifact/benchmark
% edit Makefile.inc
  - only necessary if you want to use foMPI: uncomment
    lines 11 to 15
% make
end results:


- bench_gdi.bfs (for BFS and k-hop benchmarks)
- bench_gdi.bi
- bench_gdi.cd1p
- bench_gdi.gnn
- bench_gdi.lcc
- bench_gdi.oltp.lb.lat (OLTP LinkBench workload: latency)
- bench_gdi.oltp.lb.tp (OLTP LinkBench workload: throughput)
- bench_gdi.oltp.ri.lat (OLTP read intensive workload: latency)
- bench_gdi.oltp.ri.tp (OLTP read intensive workload: throughput)
- bench_gdi.oltp.rm.lat (OLTP read mostly workload: latency)
- bench_gdi.oltp.rm.tp (OLTP read mostly workload: throughput)
- bench_gdi.oltp.wi.lat (OLTP write intensive workload: latency)
- bench_gdi.oltp.wi.tp (OLTP write intensive workload: throughput)
- bench_gdi.pr (for PageRank)
- bench_gdi.wcc

```

### Installation: Graph500

time: a few minutes

The Graph500 BFS reference implementation is used as baseline and to generate the root vertices for the BFS/k-hop benchmarks of GDI-RMA.

```
% cd GDI_artifact
% wget
  https://github.com/graph500/graph500/archive/graph500-3.0.0.tar.gz
% tar xfz graph500-3.0.0.tar.gz
% cp graph500_main.c graph500-graph500-3.0.0/src/main.c
  - modifications:
    - write Graph500 measurement data into the file
      bfs_times.txt
    - write root vertices into the file bfs_root.txt
```

```

- execute 100 BFS runs:
  - 10 unique root vertices (line 13)
  - 10 runs for each unique root vertex (line 14)
% cd graph500-graph500-3.0.0/src
% edit Makefile:
- add -fcommon to CFLAGS (line 1)
- set MPICC to the MPI C-compiler of your system (line 3)
- move '$(LDFLAGS)' after -lm (line 15)
% make graph500_reference_bfs
% cp graph500_reference_bfs ../../benchmark

end result: graph500_reference_bfs

```

## Running the Benchmarks

All executables have the same command line interface, however not every parameter is significant for every executable.

```

GDI Benchmark
-b <bsize> : block size [512]
-d : use directed edges [false]
-e <efactor> : edge factor [16]
-f <file> : load graph from file
-i <iter> : iterations for CDLP/PageRank/WCC [5]
-l <layers> : layers for GNN [5]
-m <msize> : memory size per process [4096]
-n <nverts> : number of vertices [0]
-o : vertex UIDs start at one [false]
-r <rcount> : number of queries [200]
-s <scale> : log2(# vertices) [3]
-t <time> : duration to run Linkbench queries [5]
-v <vector> : size of feature vector for GNN [500]
-w <damp> : damping factor for PageRank [0.85]
-h : print this help message

```

We used the default values of 512 Bytes for the GDI-RMA block size (-b) and 16 for edge factor (-e) of the Kronecker graphs for all experiments illustrated as figures in the article. We used a memory size (-m) of 300000000 Bytes (300 MB) for each process of the XC40 compute nodes and 900000000 Bytes (900 MB) for each process of the XC50 compute nodes on Piz Daint. The memory size describes the amount of memory that each process reserves for the graph database. -s <scale> varies the number of vertices for the Kronecker graph generation:  $2^{scale}$ . Directed edges (-d) are used for the following OLAP workloads:

- GNN
- PageRank

The other workloads (BI2, BFS, k-hop, CDLP, LCC, OLTP, WCC) use undirected edges. -f, -n and -o are only significant, when a graph is loaded from a file.

We used the default value of 5 for the number of iterations (-i) of the CDLP and WCC workloads. For PageRank we used 10 iterations and the default value of 0.85 for the dampening factor (-w). We used the default value of 5 layers (-l) for the GNN workload and varied the size of the feature vectors with -v: 4, 16, 64, 256 and 500. -r adjusts the number of queries per process to be run for the OLTP workloads. We used 10100 queries for XC40 compute nodes and 30100 for XC50 computes nodes on Piz Daint, where we used the first 100 queries of each process for warm up. By using these specific numbers, each compute node executes 360000 queries to allow for a fair comparison. The number of execution times for each query (rcount variable) of the OLAP and OLSP workloads is hardcoded in their respective main.\*.cpp file.

Before executing any benchmark, one should ensure that the LibSciBench shared library can be found by using the

following command: `export LD_LIBRARY_PATH="{PATH to installation}liblsb/build/lib:$LD_LIBRARY_PATH"`.

We provide example Slurm job scripts for OLTP workloads on Piz Daint:

- benchmark/job.oltp.lb.XC40.sh
- benchmark/job.oltp.lb.XC50.sh

bench\_gdi.bfs combines the benchmarks for BFS and k-hop, because both benchmarks use the root vertices generated by the Graph500 BFS reference implementation as the basis for their queries, since the Graph500 code ensures that the root vertices are not isolated. The idea is to first run the Graph500 BFS reference implementation, and then the GDI-RMA BFS and k-hop benchmarks during the same job. We use 2, 3 and 4 hops for the evaluation of the k-hop benchmark. The Graph500 BFS reference implementation experiences performance penalties, when the number of processes is not a power of two, so we use only eight processes (instead of twelve) per compute node for the Graph500 BFS on the XC50 compute nodes on Piz Daint.

We provide an example Slurm job script for the BFS/k-hop workload on Piz Daint: benchmark/job.bfs.sh.

Each binary generates a result file for each rank during the experimental runs. In the following, we list the prefix for each binary:

- bench\_gdi.bfs:
  - lsb.gdi\_bfs.r{rank number}
  - lsb.gdi\_k\_hop.r{rank number}
- bench\_gdi.bi: lsb.gdi\_bi.r{rank number}
- bench\_gdi.cdlp: lsb.gdi\_cdlp.r{rank number}
- bench\_gdi.gnn: lsb.gdi\_gnn.r{rank number}
- bench\_gdi.lcc: lsb.gdi\_lcc.r{rank number}
- bench\_gdi.oltp.lb.lat: lsb.gdi\_oltp.lb.lat.r{rank number}
- bench\_gdi.oltp.lb.tp: lsb.gdi\_oltp.lb.tp.r{rank number}
- bench\_gdi.oltp.ri.lat: lsb.gdi\_oltp.ri.lat.r{rank number}
- bench\_gdi.oltp.ri.tp: lsb.gdi\_oltp.ri.tp.r{rank number}
- bench\_gdi.oltp.rm.lat: lsb.gdi\_oltp.rm.lat.r{rank number}
- bench\_gdi.oltp.rm.tp: lsb.gdi\_oltp.rm.tp.r{rank number}
- bench\_gdi.oltp.wi.lat: lsb.gdi\_oltp.wi.lat.r{rank number}
- bench\_gdi.oltp.wi.tp: lsb.gdi\_oltp.wi.tp.r{rank number}
- bench\_gdi.pr: lsb.gdi\_pr.r{rank number}
- bench\_gdi.wcc: lsb.gdi\_wcc.r{rank number}

## Experiments

It won't be possible to recreate the mixed XC40/XC50 workloads with the provided code base, because for those experiments we had to alter the code base to account for the different amount of memory per process.

We estimate that it takes two weeks to run all necessary experiments presented in the evaluation section of the submission. This

time can be reduced by running fewer experiments (number of computes nodes) or by reducing the runtime of the experiments by reducing the scale of the Kronecker graph: A scale reduction by one cuts the number of vertices in half and should similarly reduce the runtime.

Figure 4a and 4c: Throughput Weak Scaling OLTP Workloads. We executed the following commands:

```

srun ./bench_gdi.oltp.{lb|ri|rm|wi}.tp -s <scale>
-m 300000000 -r 10100 (XC40)
srun ./bench_gdi.oltp.{lb|ri|rm|wi}.tp -s <scale>
-m 900000000 -r 30100 (XC50)

```

#nodes	scale
8	26
16	27
32	28
64	29
128	30
256	31
512	32
1024	33
2048	34

Table 1: Experimental parameters for Figures 4a and 4c

Figure 4b and 4d: Throughput Strong Scaling OLTP Workloads. We executed the following commands:

```

srun ./bench_gdi.oltp.{lb|ri|rm|wi}.tp -s 26
-m 300000000 -r 10100 (XC40)
srun ./bench_gdi.oltp.{lb|ri|rm|wi}.tp -s 26
-m 900000000 -r 30100 (XC50)

```

We used 8, 16, 24, 32, 40, 48, 56 and 64 compute nodes.

Figure 5a: Latency Linkbench OLTP Workload. We executed the following command:

```

srun ./bench_gdi.oltp.lb.lat -s 23 -m 300000000
-r 10100 (XC40)

```

We used 1, 2, 4 and 8 compute nodes.

Figure 6a: Weak Scaling OLAP Workloads. We executed the following commands:

```

srun ./bench_gdi.cd1p -s <scale> -m 900000000 (XC50)
srun ./bench_gdi.pr -s <scale> -m 900000000 -i 10
-d (XC50)
srun ./bench_gdi.wcc -s <scale> -m 900000000 (XC50)

```

Figure 6b: Strong Scaling OLAP Workloads. We executed the following commands:

```

srun ./bench_gdi.bi -s 26 -m 900000000 (XC50)
srun ./bench_gdi.cd1p -s 26 -m 900000000 (XC50)
srun ./bench_gdi.lcc -s 23 -m 900000000 (XC50)
srun ./bench_gdi.pr -s 26 -m 900000000 -i 10 -d (XC50)
srun ./bench_gdi.wcc -s 26 -m 900000000 (XC50)

```

We used 8, 16, 24, 32, 40, 48, 56 and 64 compute nodes.

#nodes	scale
1	23
2	24
4	25
8	26
16	27
32	28
64	29
128	30
256	31
512	32
1024	33

Table 2: Experimental parameters for Figure 6a

Figure 6c: Weak Scaling GNN Workload. We executed the following commands:

```

srun ./bench_gdi.gnn -s <scale> -m 900000000 -v 4
-d (XC50)
srun ./bench_gdi.gnn -s <scale> -m 900000000 -v 16
-d (XC50)
srun ./bench_gdi.gnn -s <scale> -m 900000000 -v 64
-d (XC50)
srun ./bench_gdi.gnn -s <scale> -m 900000000 -v 256
-d (XC50)
srun ./bench_gdi.gnn -s <scale> -m 900000000 -v 500
-d (XC50)

```

#nodes	scale
1	22
2	23
4	24
8	25
16	26
32	27
64	28
128	29
256	30
512	31
1024	32

Table 3: Experimental parameters for Figure 6c

Figure 6d: Strong Scaling GNN Workload. We executed the following commands:

```

srun ./bench_gdi.gnn -s 25 -m 900000000 -v 4 -d (XC50)
srun ./bench_gdi.gnn -s 25 -m 900000000 -v 16 -d (XC50)
srun ./bench_gdi.gnn -s 25 -m 900000000 -v 64 -d (XC50)
srun ./bench_gdi.gnn -s 25 -m 900000000 -v 256 -d (XC50)
srun ./bench_gdi.gnn -s 25 -m 900000000 -v 500 -d (XC50)

```

We used 8, 16, 24, 32, 40, 48, 56 and 64 compute nodes.

Figure 6e: Weak Scaling BFS/k-Hop Workloads. We executed the following commands:

```

srun --ntasks=<#tasks> --ntasks-per-node=8
./graph500_bfs <scale> 16
srun ./bench_gdi.bfs -s <scale> -m 900000000 (XC50)

```

#nodes	#tasks	scale
1	8	23
2	16	24
4	32	25
8	64	26
16	128	27
32	256	28
64	512	29
128	1024	30
256	2048	31
512	4096	32
1024	8192	33
2048	16384	34

**Table 4: Experimental parameters for Figures 4a and 4c**

*Figure 6f: Strong Scaling BFS/k-Hop Workloads.* We executed the following commands:

```

srun --ntasks=<#tasks> --ntasks-per-node=8
./graph500_bfs 26 16
srun ./bench_gdi.bfs -s 26 -m 900000000 (XC50)

```

We used 8, 16, 24, 32, 40, 48, 56 and 64 compute nodes. The Graph500 BFS reference implementation was only executed for a selected number of compute nodes to ensure that the total number of processes was a power of two.

#nodes	#tasks
8	64
16	128
32	256
64	512

**Table 5: Experimental Graph500 parameters for Figure 6f**

## Plotting

We provide Python scripts to visualize the experimental results in the directory `plots`. The execution of `python3 {script name}` will result in a PDF file, whose name matches the script name: `{script name}.pdf`.

*Figure 4a and 4c: Throughput Weak Scaling OLTP Workloads.*

```

plots/throughput_read_weak_scaling_bar.py (4a)
plots/throughput_write_weak_scaling_bar.py (4c)

```

- scale of the initial Kronecker graph can be updated in lines 18 and 40
- number of compute nodes used for the experiments can be updated in lines 19 and 41
- path to the result files and their naming convention can be updated in lines 21 and 43

*Figure 4b and 4d: Throughput Strong Scaling OLTP Workloads.*

```

plots/throughput_read_strong_scaling_bar.py (4b)
plots/throughput_write_strong_scaling_bar.py (4d)

```

- number of compute nodes used for the experiments can be updated in lines 18 and 38
- path to the result files, their naming convention and the Kronecker graph scale can be updated in lines 20 and 40

*Figure 5a: Latency Linkbench OLTP Workload.*

```

plots/latency.oltp.lb.gda.py

```

- path to the result files and their naming convention can be updated in line 30

*Figure 6a: Weak Scaling OLAP Workloads.*

```

plots/global_weak_scaling.py

```

- scale of the initial Kronecker graph can be updated in lines 15, 28 and 40
- number of compute nodes used for the experiments can be updated in lines 16, 29 and 41
- path to the result files and their naming convention can be updated in lines 18, 30 and 42

*Figure 6b: Strong Scaling OLAP Workloads.*

```

plots/global_strong_scaling.py

```

- number of compute nodes used for the experiments can be updated in lines 19, 30, 40, 50 and 60
- path to the result files, their naming convention and the Kronecker graph scale can be updated in lines 21, 31, 41, 51 and 61

*Figure 6c: Weak Scaling GNN Workload.*

```

plots/gnn_weak_scaling.py

```

- scale of the initial Kronecker graph can be updated in line 16
- number of compute nodes used for the experiments can be updated in line 17
- path to the result files and their naming convention can be updated in lines 20, 29, 38, 47 and 56

*Figure 6d: Strong Scaling GNN Workload.*

```

plots/gnn_strong_scaling.py

```

- number of compute nodes used for the experiments can be updated in line 16
- path to the result files, their naming convention and the Kronecker graph scale can be updated in lines 19, 28, 37, 46 and 55

*Figure 6e: Weak Scaling BFS/k-Hop Workloads.*

```

plots/bfs_khop_comb_weak_scaling.py

```

- scale of the initial Kronecker graph can be updated in lines 16, 29 and 41
- number of compute nodes used for the experiments can be updated in lines 17, 30 and 42
- path to the result files and their naming convention can be updated in lines 19, 31 and 45

*Figure 6f: Strong Scaling BFS/k-Hop Workloads.*

plots/bfs\_khop\_comb\_strong\_scaling.py

- number of compute nodes used for the experiments can be updated in lines 17, 28 and 40
- path to the result files, their naming convention and the Kronecker graph scale can be updated in lines 18, 29 and 41