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

Maciej Besta[1*#], Robert Gerstenberger[1*#], Marc Fischer[2], Michał Podstawski[3,4], Jürgen Müller[5], Nils Blach[1], Berke Egeli[1], George Mitenkov[1], Wojciech Chlapek[6], Marek Michalewicz[7], Torsten Hoefler[1*]

[1]ETH Zurich; [2]PRODYNA (Schweiz) AG; [3]Warsaw University of Technology; [4]TCL Research Europe; [5]BASF SE; [6]ICM UW; [7]Sano Centre for Computational Medicine; [*]Corresponding authors, [#]alphabetical order

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.

Website, code, and the GDI specification:
http://spcl.inf.ethz.ch/Research/Parallel_Programming/GDI

1 INTRODUCTION

Graph databases (GDBs) enable storing, processing, and analyzing large and evolving irregular graph datasets in areas as different as medicine or sociology [41, 156]. GDBs face unique design and compute challenges. First, GDB datasets are huge *and* complex. While they can have over tens of trillions of edges [140], 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[1]. 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 [138]. *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 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) [160] 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 [41] (e.g., Neo4j [186], Apache TinkerPop [1], or JanusGraph [213]) 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.* Besides providing some advice to implementors, GDI does not specify any architecture- or implementation-specific details, instead focusing on a clear specification of the semantics of its routines. 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 offer large amounts of total memory (even up to 10PB [57]), compute power (even more than 40 million cores [57]), and network bandwidth (even 1.66 PB/s and 46.1 TB/s for the injection and bisection bandwidth, respectively [3]). Hence, they can accommodate the enormous requirements posed by GDB datasets, 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 [31, 34, 92, 92, 110, 113, 122, 123, 144, 145, 158, 174, 181, 192, 199, 225, 229]. RDMA is widely supported on DM systems, both in the cloud infrastructure (e.g., on Azure [127],

[1]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 [53]	🔋	✖	✖	✖	🔋	✖	✖	✖	245	2,940	3.2 TB	6.2B	3.7B	✖	✖	✖	✖	✖	128 GB	✖
GAIA [179]	✖	✖	✖	✖	✖	🔋	✖	✖	✖	✖	✖	✖	✖	16	384	1.96 TB	17.79B	2.69B	512 GB	✖
G-Tran [66]	🔋	✖	✖	🪫	🔋	✖	✖	✖	10	160	*1.28 TB	0.495B	0.082B	✖	✖	✖	✖	✖	128 GB	🪫
Neo4j [188]	✖	✖	✖	✖	🔋	🔋	🪫	🔋	1	64	1.6 TB	17.79B	2.69B	1	64	1.6 TB	17.79B	2.69B	2 TB	✖
TigerGraph [218]	✖	✖	✖	✖	🔋	🔋	✖	🔋	40	1600	17.7 TB	533.5B	72.62B	36	4,608	N/A	539.6B	72.6B	1 TB	✖
JanusGraph [214]	✖	✖	✖	✖	🔋	🪫	✖	🔋	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	✖
Weaver [82]	✖	✖	✖	✖	🔋	✖	✖	✖	44	352	0.976 TB	1.2B	0.08B	✖	✖	✖	✖	✖	16 GB	✖
Wukong [198]	🔋	✖	✖	✖	🔋	✖	✖	✖	6	120	*0.384 TB	1.41B	0.387B	✖	✖	✖	✖	✖	64 GB	✖
ByteGraph [138]	✖	✖	✖	✖	🔋	🔋	🔋	✖	10	160	N/A	N/A	N/A	130	N/A	113 TB	N/A	N/A	1 TB	🪫
This work	🔋	🔋	🔋	🔋	🔋	🔋	🔋	🔋	**7,142**	**121,680**	**77.3 TB**	**549.8B**	**34.36B**	**7,142**	**121,680**	**36.5 TB**	**274.9B**	**17.2B**	**64 GB**	🔋

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. *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.*

Alibaba [5], or Oracle [169] data centers), in general Ethernet networks (with RoCE [111] or iWARP [96]), and in the HPC networks (e.g., on Cray [86], IBM [16], or InfiniBand [212]). Importantly, the RDMA API in most of these systems can be abstracted into a generic *Remote Memory Access (RMA)* programming model [92]. Our implementation of GDI, called *GDI for Remote Memory Access (GDI-RMA)*, facilitates portability on such systems as it uses this generic RMA API.

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)* [63, 107] to deliver scalable transactions involving many processes (e.g., for large-scale OLAP queries) with well-defined semantics. Collectives are a form of high-performance data exchange that has been tuned over many years, resulting in schemes that are provably optimal or near-optimal (in minimizing the communicated data) and that enable efficient use of all compute resources [107, 200, 210].

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 [9, 17]. We use established problems such as BFS [42, 170] and state-of-the-art workloads such as Graph Neural Networks [36, 51, 94, 99, 132, 191, 230, 236, 237]. 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, outperforming the JanusGraph modern graph database by up to orders 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)** [41] (also called the property graph [8]). LPG is a primary data model used in many GDBs, including the leading Neo4j GDB [41]. 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 $|E| = m$. An edge $e = (u, v) \in E$ is a tuple of two vertices, where u is the out-vertex (origin) and v is the in-vertex (target). If G is undirected, an edge $e = \{u, v\} \in E$ is a set of two vertices. 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 [9, 17] identify two main classes of graph database workloads targeting the LPG graph model: *interactive workloads* [85] (mostly OLTP) and *graph analytics* [112] (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) [205, 206] which fetch large parts of a graph and often use data summarization and aggregation. They are sometimes referred to as *Online Serving Processing* (OLSP) [138]. 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. Thus, higher-level parts of a graph database (query methods, query planer, execution engine, etc.) can run vendor agnostic. 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 [29]. 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 [97]. 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 it 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. Its goal is to store the data in a reliable way and provide fast data access.

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.

As an example, consider the following query: "*How many people are over 30 years old and drive a red car?*". A corresponding Cypher query by the user could be as follows: `MATCH (per:Person) WHERE per.age > 30 AND per-[:OWN]-> vehicle(:Car) AND vehicle.color = red RETURN count(per)`. The mid-layer decides on the necessary steps, which would be retrieving "Person" vertices (possibly using an index), checking the age condition, retrieving edges with the "OWN" label, checking the adjacent vertices for the "Car" label, verifying if the found "Car" vertices are red. If an index is available that is not only associated with the label 'Car' but also the property type 'color', the query planer might instead decide to start at the other side of the data flow and follow-up in the opposite direction. That would be a logical step under the assumption, that there are fewer red-painted cars than persons in the database, so the amount of initial objects would be (much) smaller. However in the following paragraphs illustrating the different layers, the first scenario is assumed. Next, the mid-layer runs these steps, coordinating each process to examine their local portions of "Person" vertices and to check whether they fulfill all of the conditions set out above. The sum of the vertices can be accumulated locally and the global result is computed with a global reduce operation. To execute the above, the mid-layer performs a local index query to retrieve all the vertices with the label "Person", and translate the condition steps of the query planner into a series of function calls to the storage engine. For example, to check the age condition, one would first retrieve the "age" property of the respective vertex, and afterwards check whether that property fulfills the age condition. For the edge retrieval, it is possible to let the storage handle the filtering with the use of a specifically crafted so-called constraint object. The storage engine layer receives the function calls from the mid-layer, and it handles the retrieval of data from the intermediate data representation of the (possibly distributed) storage backend layer. Finally, the storage backend directly accesses physical storage as instructed by the storage engine.

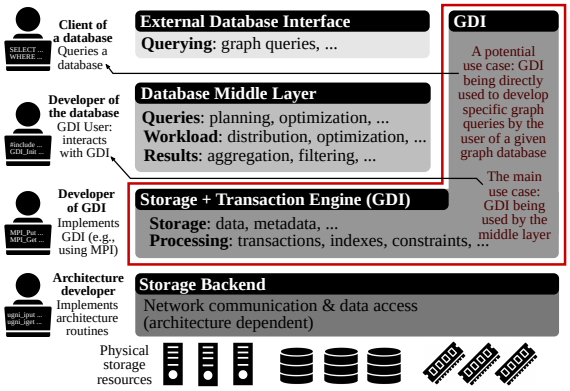


Figure 1: GDI with respect to other parts of the graph database landscape.

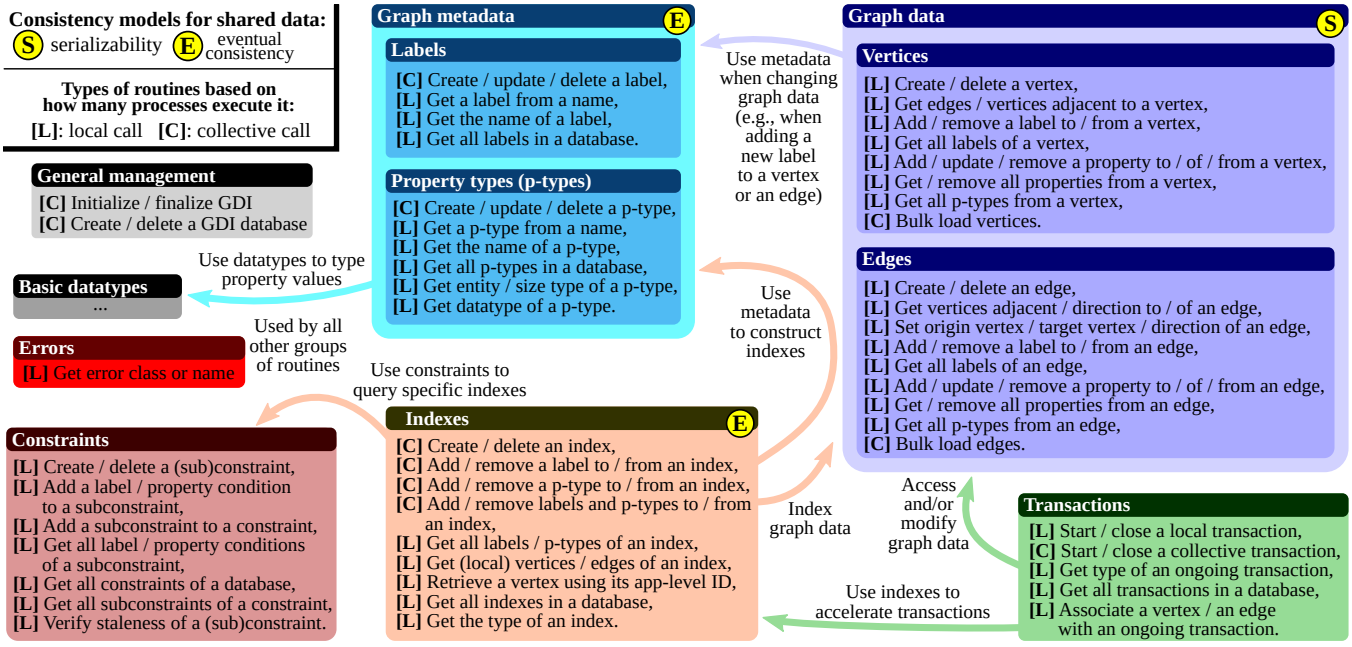


Figure 2: Illustration of the classes of GDI routines.

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 [63, 107]. 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 [107, 108]. They also foster portability and programmability [108] 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 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.

Atomicity ensures that the operations are treated as a single unit and either all succeed or completely fail. Consistency ensures that before and after a transaction, the database is always in a consistent state. Isolation ensures that concurrent transactions behave as if they were run in some sequential order. Durability ensures that the effects of a committed transaction remain in the database even in the case of a system failure.

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.

GDI differentiates between **transaction critical** and **transaction non critical errors**. If a function returns a transaction critical

error, the transaction is guaranteed to fail. GDI does not offer functions to retry a transaction or to recover from a transaction critical error: The user must start a new transaction.

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, detailed in Section 5, 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*. These handles can be passed as arguments, and they can participate in assignments and comparisons. Each vertex and edge handle must be instantiated with an **association**. 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. Opaque objects and their handles are process specific, and they are only significant at their respective allocating process and cannot be shared with other processes via communication.

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.

GDI intends the allocation and deallocation of objects to appear to the user as if the information for those objects were copied, so that semantically opaque objects are separate from each other. However, this does not mean that GDI implementations may not employ optimizations such as references and reference counting, if they are hidden from the user.

3.6 Flexible Indexes & Constraints

It is desirable for a graph databases to enable a fast lookup of vertices and edges *satisfying certain application-specific conditions*, for example having a specific label or a property. Here, GDI provides an interface for indexing structures that can be explicitly used by the GDI user. We enable this interface to have flexible indexing conditions, to support a wide variety of graph database workloads. Explicit indexes are queried by using boolean formulas (referred to as **constraints** in GDI) in disjunctive normal form (DNF). GDI constraints support arbitrary conditions regarding labels and properties. Explicit indexes impose no restrictions on what indexes the implementation of GDI might use internally.

3.7 Performance-Centric Syntax & Semantics

GDI fosters many optimizations by enabling the GDI user to provide GDI with extensive (but optional) information about their setting. As an example, we discuss property types. First, one can inform GDI whether there can be at most one property entry of a given property type, on a given single vertex or edge, or whether there may be multiple entries of this property type on one vertex/edge. Next, one can inform GDI about the datatype of the elements of the property value. Finally, one can also explicitly specify whether the property type has a certain size limitation, or if it has a fixed size.

3.8 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 [186]. In GDI, we clearly specify it.

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

3.9 Multiple Parallel Databases

GDI supports running multiple concurrent distributed GDBs in a single environment. For this, GDI functions accept an optional handle to a graph database object. While we do not currently focus on this scenario, we decided to include it, predicting that it will facilitate future integration of GDI with cloud settings where multiple users run their separate databases.

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. Note that these are general recommendations; GDI users are free to use any transaction type for any workload, depending on their domain knowledge and insights.

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.

We show how to use GDI to develop the above-mentioned workloads. 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_FRIENDOF */) {
9   GDI_GetVerticesOfEdge(&v_originID, &v_targetID, eH); //
       Retrieve target vertex
10   neighborsID.add(v_targetID) /* add target vertex to
       neighborIDs 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 [36, 230]. 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). For clarity, we present a simplified query with the most important communication-intense operations.

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.

5.1 RDMA & RMA, and How We Use It

RDMA has become widely used thanks to the high availability of RDMA-enabled network interface cards (RNICs). RDMA has been used with modern database management systems to accelerate data replication [54, 116, 130, 207, 235], distributed transactions [80, 227, 228, 234], distributed index structures [239], general query processing [45, 187], and analytical workloads [21, 139]. 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 *put*s and *get*s that – respectively – write to and read from remote memories. Puts/gets offer very low latencies, often outperforming message passing [91]. One can also use remote *atomics* [32, 103, 160, 193]; here, we additionally harness **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 [2]. All these routines are supported by virtually any RDMA architecture, facilitating a wide portability of GDA.

```
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 "cnsrt" 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 (see explanation of the Cypher query in 3.1): "MATCH (per:Person) WHERE per.age>30 AND per-:OWN->vehicle(:Car) AND vehicle.color = 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).

Fully offloaded one-sided RDMA usually outperforms two-sided communication both in RDMA and in message passing. Yet, it also entails a complex and involved design, as conflicting accesses must be fully resolved by sending processes. To alleviate this, we spent a significant effort on ensuring that GDA is programmable, by making it highly modular and easy to reason about.
The semantics of the used operations are as follows:

`GET(local, remote), PUT(local, remote),`

`CAS(local_new, compare, result, remote)`

where `GET` fetches a value from a remote address `remote` into a local variable `local`, `PUT` insert the value of a local variable `local` into a remote address `remote`, and `CAS` compares a value at a remote address `remote`, and if it equals `compare`, it replaces it with `local_new`, saving the previous value at `remote` into a local variable `result`). Atomic variants of put/get are indicated with `APUT/AGET`.

5.2 Overview of GDI-RMA Design

We now overview GDI-RMA (GDA), see Figure 3 (⓪). We will detail the concepts mentioned here in the following subsections. Our implementation[2] 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 GDI and GDA). The most important modules are management structures for GDI and parallel databases (Ⓖ), metadata structures (Ⓜ), indexes and associated constraints (Ⓘ, Ⓒ), state of collective and local transactions (Ⓣc, Ⓣl), and graph data (Ⓓ). Structures that are small or independent of #vertices and #edges (Ⓖ, Ⓜ, Ⓣc) are replicated on each process to foster simplicity and high performance. All other structures are sharded[3]. This, combined with our fully distributed transactions, enables fast and scalable processing of very large graph datasets, being limited only by the cluster size.

[2]We use the foMPI implementation of MPI One-Sided routines [92].
[3]Partitioning of data onto separate servers.

5.3 Graph Data

The central part of GDA is associated with graph data (Ⓓ 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 LL routines form a clean and graph-centric API; any performance optimizations can be done under the hood at the BGDL level.*

Accessing Graph Data. As an example, we discuss accessing a selected vertex v and its property, see the top part of Figure 3. 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 [2]. Then, ***vID*** is used to construct a handle ***vH***, which is a pointer to v in the local memory of a calling process. Handles are typed and the object they point to has to match that type. Finally, while one can access any label or property of v directly in v's holder, fetching additional information about the given metadata can be done using label/property ***integer IDs*** (used only within GDA).

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

5.4 Logical Layout Level

We shard the graph data (vertices, edges, and their associated labels and properties) across all processes.

5.4.1 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

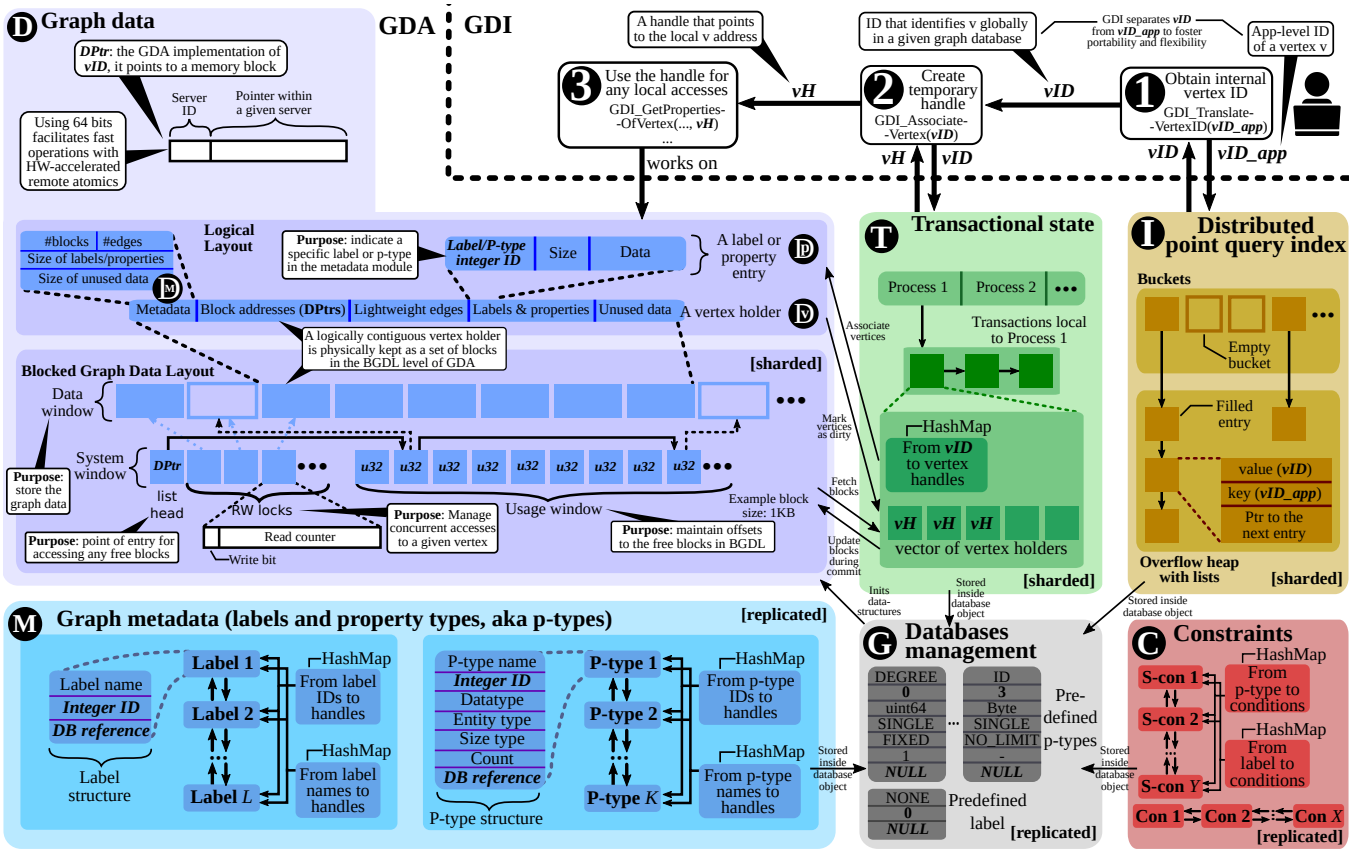


Figure 3: Details of the GDI-RMA (GDA) implementation, and its interaction with GDI. In the upper part of the figure, we illustrate a simple sequence of steps (taken within a transaction) to access a selected property of a given vertex v.

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.

5.4.2 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.

An edge UID is a data structure that identifies a lightweight edge uniquely in the database. The data structure consists of two parts: a vertex UID and an offset for the edge data structure of the vertex identified by the vertex UID. Therefore the same edge can be identified by two different edge UIDs, depending on which vertex is used as a base. An edge UID takes up 12 bytes of memory, where the lower 8 bytes contain the vertex UID and the upper 4 bytes the offset to the edge.

5.4.3 Labels & Properties. GDA uses a data structure called a *label* or *property entry* to store a single label or property. GDA treats labels internally as properties for storage. The ***integer ID*** serves two purposes: it indicates whether an entry is unused/empty (value 0) or whether it is the last entry (value 1), and to store the integer ID of a given label/p-type (value 2 for a label, any other value for a specific p-type).

5.5 Blocked Graph Data Layout (BGDL) Level

The graph data (vertices and edges with their labels and properties) is mapped to fixed-size blocks. The block size is specified by the user, enabling a tunable tradeoff between communication amount and memory consumption. Specifically, the larger the blocks are, the less communication may be needed to conduct DM graph operations (because fewer blocks are required to cover given graph data), but also the more memory may be used, as larger parts of blocks may be unavailable (as in internal fragmentation). This tradeoff can be tuned by the user separately for different applications. If the size of a memory block is big enough to contain all four memory segments, only one memory block is used. The first memory block of a vertex is called the primary block.

❗ **Major Design Choice & Insight**: *Using fixed-size blocks in BGDL 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.

Handling blocks is done internally and is not accessible through the library interface. The two basic operations are

```
void acquireBlock( int target_rank, DPtr* dp ),
void releaseBlock( DPtr* dp ).
```

`acquireBlock` tries to allocate a block on process `target_rank`. `releaseBlock` frees a specified block. The block layer is oblivious to the actual contents of the blocks.

We use windows for the RMA based implementation of blocks. There are three windows: the *data* window, the *usage* window, and the *system* window. The data window contains all the blocks that constitute the vertex and edge holder objects. Management of the blocks is done with the usage and the system windows. The usage window is essentially an indexing structure pointing to free blocks in the general window. It holds a linked list, where each link element contains the offset for the next free block in the data window. Finally, the system window contains the pointer to the first unused element in the linked list of the usage window as well as the locks that correspond to the blocks stored on this process.

In block management, for highest performance, we use non-blocking communication and lock-free synchronization. We now sketch the `acquireBlock` routine, `releaseBlock` is similar. First, the head of the list maintained by the usage window is retrieved with a get from the system window. Second, the index of the next free block in the data window is obtained with another get from the element that list head points to. Third, we use an atomic CAS to replace the index of the current free block in the list head with the index of the next free block. If the operation is successful, no other process obtained a free block from the target process in the time between the first get operation in step 1 and the issued CAS. In this case, a distributed pointer with the rank of the target process and the free block offset is created and returned to the origin process. If the CAS is unsuccessful, the whole operation restarts at step 2 with the information just obtained from the CAS. If the get in step 1 retrieves a NULL element, a NULL handle is returned to the origin process to indicate that the target process has no free blocks.

Block operations are prone to the ABA problem, because the list head is used to check whether anything in the list has changed, while trying to complete an acquire or release operation. GDA alleviates this with the established tagged pointer technique.

5.6 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). The state of a single process transaction is stored on its associated process, while the state of a collective transaction is replicated on each process for performance reasons. All changes applied in a transaction are visible only locally. If the transaction commits, data changes are written to the remote blocks and any associated indexes.

In order to ensure fast starts, commits, and aborts of transactions, the operations on blocks performed under the hood must be highly efficient. This includes verifying whether a transaction has already fetched blocks corresponding to a certain vertex or edge holder object, tracking of dirty blocks to be updated or removed, and deallocation of blocks.

To achieve high performance of these operations, a database maintains a list of transaction handles, with each handle holding the address of the state of a given transaction. Each such state holds two hashmaps to efficiently lookup handles of vertex and edge holder objects associated with a given transaction, given the vertex/edge internal IDs. Moreover, the transaction state also holds a list with pointers to any local and remote blocks used in the given transaction. When committing a transaction, either all dirty blocks are written back or none, meaning that no elements must be removed from the according list. Hence, we use a vector to keep track of the dirty blocks that need to be written back during a commit phase. This enables adding a new block address in $O(1)$ amortized time and destroying a whole vector in $O(1)$ time. A linked list would instead require $O(n)$ for deconstruction (amortized: $O(1)$).

Major Design Choice & Insight: *Fast intra-transaction block management is key for high-performance transactions. It can be achieved with using both hashmaps and linked lists for keeping track of any blocks used within a transaction.*

The user does not directly touch any block management. Instead, any data access and modifications are conducted via with vertex and edge holder objects. These objects and their associated routines handle the details of data caching and fetching (e.g., to prevent double fetching of the same blocks).

Similarly to the graph related data, we also keep track of the association between each transaction and its associated graph database, to enable concurrent database instances.

Scalable Reader-Writer Locking for ACI. 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. The lock semantics are standard for RW locking, i.e., no write lock is in place after the read lock is acquired, and no read locks are in place after the write lock is acquired.

5.7 Lock-Free Internal Indexing

GDA internally uses a distributed hashtable (DHT) to resolve different performance-critical tasks conducted under the hood, such as mapping application vertex IDs to internal GDA IDs. For highest performance and scalability, GDA's DHT is *fully-offloaded*, i.e., it only uses one-sided communication, implemented with RDMA puts, gets, atomics, and flushes. Its design is lock-free, it incorporates sharding, and it uses distributed chaining for collision resolution. To the best of our knowledge, this is the first DHT with all its operations being fully offloaded, including deletes. The DHT consists of a table (to store the buckets) and a heap (to store linked lists for chained elements). Each table entry consists of a key-value pair that forms a bucket, followed by a distributed pointer to a linked list of entries in the heap. Figure 3 (❶) shows the basic design of the DHT. The pseudocode is shown in Listing 4.

Insert First, a distributed pointer is fetched from the bucket (line 4) and used as the next pointer of the entry to be inserted. Then, an atomic CAS is used to introduce the entry into the linked list (line

```
1 insert(k, v) { // k: key, v: value
2  bucket = hash(k); entry = alloc(); entry.k = k; entry.v = v;
3  do {
4   AGET(next_ptr, bucket); entry.next_ptr = next_ptr;
5   CAS(&entry, next_ptr, result, bucket);
6  } while(result != next_ptr); }
7
8 lookup(k) { // k: key
9  bucket = hash(k); AGET(next_ptr, bucket);
10 if(next_ptr == NULL) return <false, NULL>;
11 do {
12  AGET(entry, next_ptr); // fetch the next ptr
13  if(entry.next_ptr == next_ptr) {
14   // entry is about to get deleted, so restart
15   return lookup(k); }
16  if(entry.k == k) return <true, entry.v>;
17  next_ptr = entry.next_ptr;
18 } while( next_ptr != NULL );
19 return <false, NULL>; }
20
21 delete(k) { // k: key
22  bucket = hash(k); AGET(next_ptr, bucket);
23  if(next_ptr == NULL) return false;
24  // deleting the first entry is similar to the code below
25  // prev_ptr is set correctly by now
26  do {
27   AGET(entry, next_ptr);
28   if(entry.next_ptr == next_ptr) {
29    // entry is about to get deleted, so restart
30    return delete(k); }
31   if(entry.k == k) {// found entry in question
32    CAS(next_ptr, entry.next_ptr, result, next_ptr+2);
33    if(result != entry.next_ptr) {
34     // either entry is about to be deleted by someone else
35     // or the next entry got deleted, so restart
36     return delete(k); }
37    CAS(entry.next_ptr, next_ptr, result, prev_ptr+2);
38    if(result == next_ptr) { dealloc(next_ptr); return true; }
39    else { // previous entry is about to be deleted
40     // remember pointer to the next entry
41     return delete(k, entry.next_ptr); } }
42    prev_ptr = next_ptr; next_ptr = entry.next_ptr;
43  } while(next_ptr != NULL);
44  return false; }
```

Listing 4: Distributed hash table operations used for internal indexing in GDA

5). If CAS fails, then another insert or delete happened concurrently, and the operation has to start again.

Lookup In general, the operation follows the bucket linked list to find the entry with the key in question. First, it fetches the distributed pointer from the bucket (line 9), to see if there are any entries in that bucket (line 10). If there are, each entry is fetched (line 12) and checked whether it contains the key in question (line 16). If the next pointer points to itself, it indicates that the entry is being deleted (line 13). In such a case, the lookup restarts (line 15).

Delete The general loop is very similar to lookup, except that an additional distributed pointer to the previous entry in the linked list is kept. The difference between the two functions occurs, once the key in question is found. Two CASes are necessary to delete a single entry. The first CAS changes the next pointer of the entry to be deleted, so it points to the entry itself (line 32). This marks that entry as ready for deletion, in case of concurrent lookups. If the 1st CAS fails (line 33), then either another process is trying to delete the entry in question and has won that race, or the following entry in the linked list was just deleted. In such a case, delete restarts (line 36). If the 1st CAS succeeds, the 2nd CAS tries to bypass the entry to be deleted by pointing the next pointer of the previous entry to the succeeding entry after the entry to be deleted (line 37). If the CAS succeeds, the entry is deallocated (line 38) and the function returns. An unsuccessful 2nd CAS indicates that the previous entry is about be deleted as well, so we restart, but retaining the original next pointer to the succeeding entry in the linked list (line 41).

❗ **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.*

5.8 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.*

5.9 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* (i.e., independence of GDA's performance of various architecture details). For this, we use the **work-depth (WD) analysis**. The WD analysis enables bounding run-times of parallel algorithms. 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 [44, 47].

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).

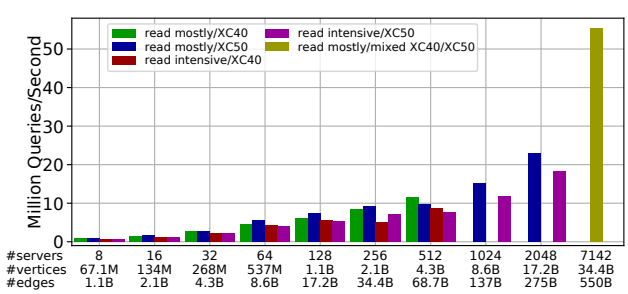
6 EVALUATION

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

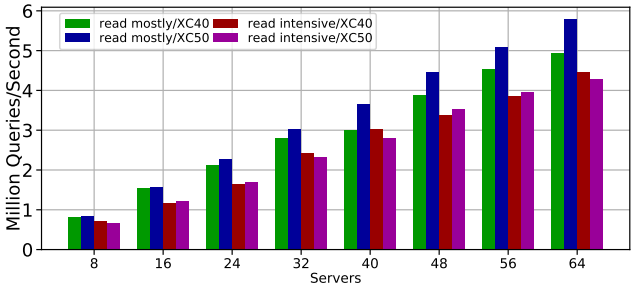
6.1 Experimental Setup, Workloads, Metrics

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 [106].

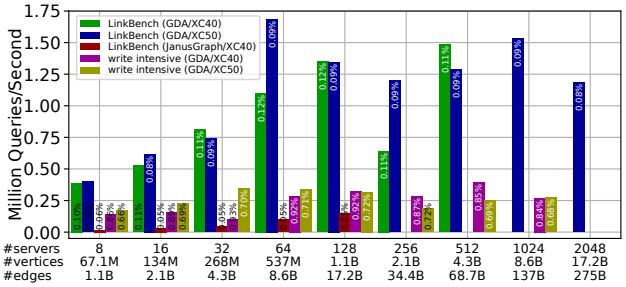
As **computing architectures**, we use the Piz Daint Cray supercomputer installed in the Swiss National Supercomputing Center (CSCS). Piz Daint hosts 1,813 XC40 and 5,704 XC50 servers. Each XC40 server has two Intel Xeon E5-2695 v4 @2.10GHz CPUs (2x18 cores, and 64 GB RAM). Each XC50 server comes with 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



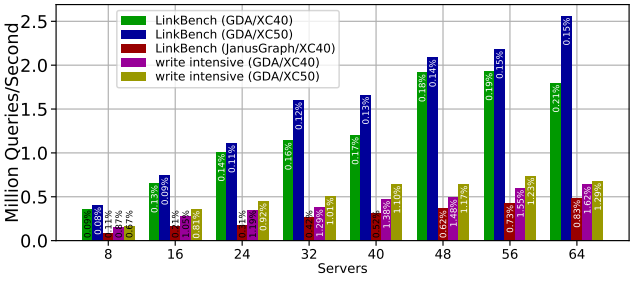
(a) Read Intensive, Read Mostly; weak scaling.



(b) Read Intensive, Read Mostly; strong scaling.



(c) LinkBench, Write Intensive; weak scaling.



(d) LinkBench, Write Intensive; strong scaling.

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).

Dragonfly topology [86, 131]. We use **full parallelism**, i.e., we run algorithms on the maximum number of cores available.

For evaluated **workloads**, we selected all three main classes of queries as specified by the LDBC and LinkBench benchmarks [9, 17] and described in Section 4, namely interactive queries [85], graph analytics [112], and business intelligence queries [205].

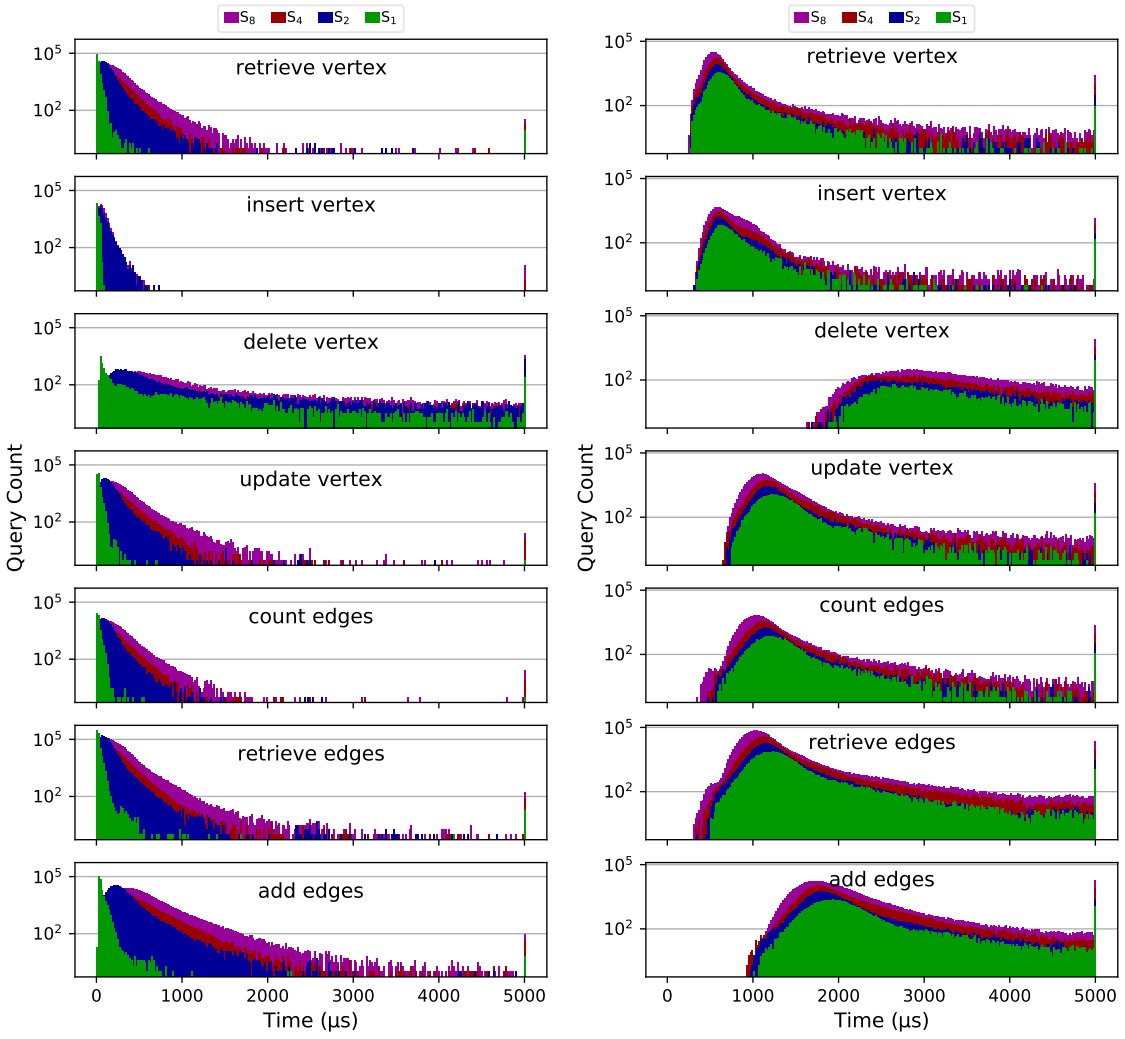
We consider three **metrics**: **latency** (i.e., how fast a query finishes), **throughput** (i.e., how many queries can we execute per time unit), and **scale**. For scale, we (1) increase the number of servers *together with* the size of the dataset (the so called "**weak scaling**") and (2) increasing the number of servers *for a fixed* dataset (the so called "**strong scaling**"). The former evaluates how GDA enables handling the increasing sizes of datasets, while the latter illustrates how GDA is able to process the existing graphs faster.

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 OLSP and OLAP queries. After an extensive investigation and configuration effort, we were able to successfully configure and use JanusGraph [213]. *This system is one 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. Even the largest publicly available graph, Web Data Commons [226], only has 128B edges and no labels/properties. Moreover, the LDBC data generator experienced regular OOM problems, whenever we tried to generate graphs of very large scales. Another problem with both real-world graphs and with the graphs generated by the LDBC generator is that they are stored in the distributed filesystem storage. Loading such large graph from disks is time-consuming and stresses compute budget. 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 [161] that uses the realistic Kronecker random graph model with a heavy-tail skewed degree distribution [137]. In this model, one specifies two input parameters, the graph vertex scale s and the edge factor e. The resulting graph has 2^s vertices and approximately e edges per vertex, for the total of $e2^s$ edges. 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



(a) GDA. **(b)** JanusGraph.

Figure 5: Details (histograms) on the latency of individual operations of the OLTP LinkBench workload. S_x indicates the data from a specific number of servers.

following analyses (we also experiment with varying these values). Internally, the vertices are distributed round robin, which is an implementation detail that can be modified by the GDA developer.

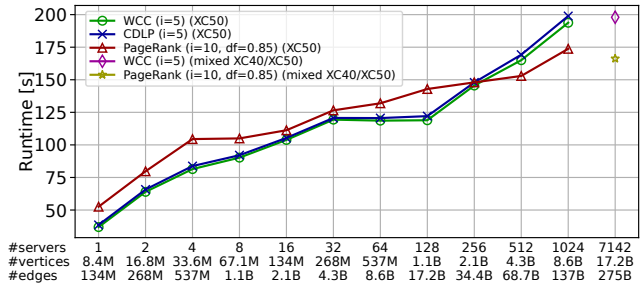
6.4 Analysis of OLTP Workloads

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 [17] and on other past GDB evaluations [66, 82], see Table 3 for details.

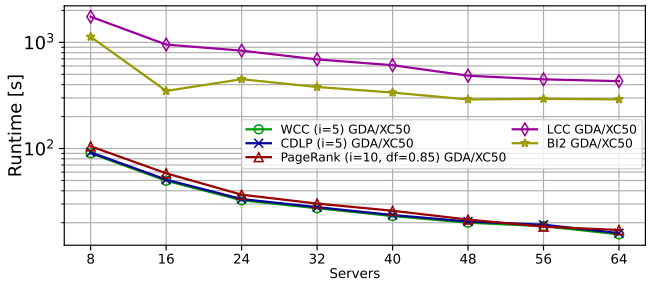
We first evaluate the overall throughput, see Figure 4. GDA achieves high scalability: adding more servers consistently improves the throughput in both strong and weak scaling. Throughput increase is particularly visible in the RI and RM workloads with more read queries, because LB and WI workloads come 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

Operation	"Read Mostly" (RM) [82]	"Read Intensive" (RI) [82]	"Write Intensive" (WI) [66]	LinkBench (LB) [17]
Read queries:	**99.8%**	**75%**	**20%**	**69%**
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%
Update queries:	**0.2%**	**25%**	**80%**	**31%**
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%

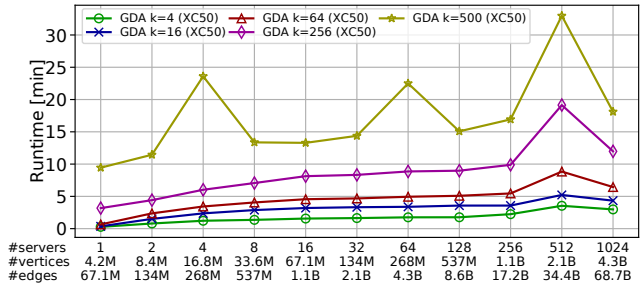
Table 3: OLTP workloads described in this paper. We varied the fractions of specific types of operations for broad investigation beyond the ones provided here, all results followed similar patterns to those described here.



(a) PR, CDLP, WCC; weak scaling.

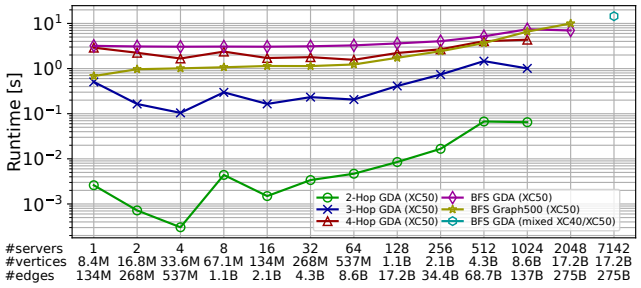


(b) PR, CDLP, WCC, LCC, BI2; strong scaling.

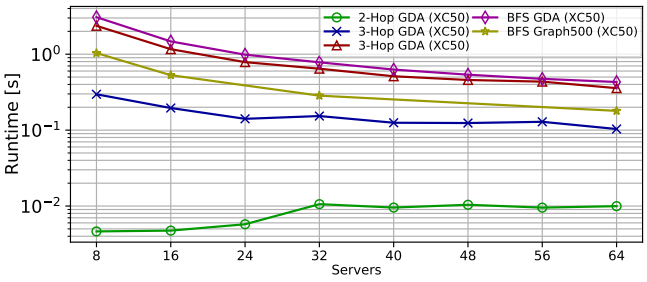


(c) GNN; weak scaling.

(d) GNN; strong scaling.



(e) BFS, k-hop; weak scaling.



(f) BFS, k-hop; strong scaling.

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.

bandwidth per core. Moreover, we note that very low percentages 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 1μs (for 1 server) and close to 10–100μs (for more servers), even for demanding vertex deletions. JanusGraph requires at least 500μs for all the operations (in most of cases), with no operation being faster than 200μs, even for the single server scenario. Vertex deletions start at around 2000μs. 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.

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 LDBC SNB (BI2) [206], and Graph Neural Networks (GNN; training of the graph convolution model [132]). 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).

We compare GDA to a very competitive target, the Graph500 implementation of BFS [161]. It is a highly 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.

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 are 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 [134] and WebGraph [48] 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 ≈3.56 billion vertices and ≈128 billion edges in ≈15s 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 ≈3× while #servers increase by 3.49×.

One recent study with large-scale executions is from the commercial ByteGraph system [138]. 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) [218], their servers have significantly more memory (each has ≈1TB 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 PORTABILITY DISCUSSION

Our work fosters two types of portability. First, we incorporate a similar type of portability as MPI. Namely, if a given graph database implementation adheres to the GDI specification, and if there is an available implementation of GDI for a given architecture, then the implementation would seamlessly compile and execute. Further, GDI is fully independent of the architecture details and it thus can be seamlessly implemented on x86, ARM, and others. While facilitating GDI's adoption, this would still require separate GDI implementations for different RDMA architectures, e.g., Cray, IBM, or InfiniBand. To alleviate this, we use plain RMA operations (puts, gets, atomics) to implement GDI-RMA. Such operations are broadly supported by RDMA networks. This facilitates porting the GDI-RMA code to other RDMA settings.

8 RELATED WORK

GDBs have been researched in both academia and industry [10, 11, 75, 90, 100, 124, 133], in terms of query languages [7, 8, 49, 208], database management [49, 120, 156, 172, 175], compression and data models [23, 35, 40, 43, 146, 147, 162], execution in novel environments such as the serverless setting [71, 150, 219], and others [77]. Many graph databases exist [6, 12–15, 46, 55, 56, 59, 65, 73, 78, 82, 87, 89, 125, 126, 151, 154, 155, 164–168, 178, 182, 183, 202, 209, 211, 214, 217, 218, 233, 238, 240]. 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 [85], LinkBench [17], and BG [20]), OLAP workloads (Graphalytics [112]), or business

intelligence queries (BI [205, 206]). One can express these workloads using portable and programmable GDI building blocks. Moreover, there are many evaluations of GDBs [58, 69, 79, 118, 142, 143, 153, 216]. We complement these works by providing a large-scale RDMA-focused evaluation.

RDMA & RMA RDMA has been one of the enablers of high performance on supercomputers and data centers, and is widely supported [5, 16, 96, 111, 127, 169, 212]. Numerous libraries and languages offer RMA features. Examples include MPI-3 RMA [160], UPC [222], Titanium [104], Fortran 2008 [114], X10 [64], or Chapel [61]. We target RDMA and RMA systems with our implementation.

Resource Description Framework (RDF) [135] is a standard to encode knowledge in ontological models and in RDF stores using triples [101, 159, 171]. There exists a lot of works on RDF and knowledge graphs [4, 18, 67, 72, 121, 141, 141, 177, 184, 185, 196, 197, 203, 220, 223, 224, 232]. We focus on graph databases built on top of LPG, and thus RDF designs are outside the scope of this work.

Dynamic, Temporal, and Streaming Graph Frameworks There are many systems that process dynamic, temporal, and streaming graphs [27, 68, 189]. Their design overlaps with graph databases, because they also focus on high-performance graph queries and updates, and on solving global analytics. Examples include Betweenness Centrality [83, 149, 176, 201, 221], Graph Traversals [33, 39, 52, 83, 128, 148, 194, 195], Connected Components [83, 88, 88, 95, 148, 195], Graph Coloring [24, 81], Matchings [22, 28, 163], and many others [38, 42, 76, 84, 93, 102, 115, 136, 215]. Recent efforts work towards processing subgraphs [180] such as maximal cliques or dense clusters [25, 26, 40, 60, 70, 109, 117]. However, such frameworks usually do not focus on rich data, do not use the full LPG model, and rarely consider ACID properties, thus being outside the focus of this paper.

Graph Processing APIs There are efforts for developing APIs different parts of the graph processing landscape. Importantly, GraphBLAS is a standard for formulating graph algorithms using linear algebra building blocks [128, 129]. It targets shared-memory CPU environments (GraphMat [152, 204]), shared-memory GPU settings (GraphBLAST [231]), and distributed-memory environments (CombBLAS [19] and others [50]). The difference is that these APIs focus on analytics on basic graph models (simple graph/multi-graph/weighted graphs) and on incorporating sparse linear algebra as the main building blocks. GraphBLAS routines could be used together with GDI. For example, the OLAP analytics supported by GDI could be implemented using GraphBLAS codes.

Graph Learning, Graph Neural Networks, Neural Graph Databases Graph neural networks (GNNs) have recently become an established part of the deep learning landscape [30, 36, 51, 62, 74, 94, 98, 99, 119, 157, 173, 190, 191, 230, 236, 237]. The versatility of GNNs brings a promise of enhanced analytics capabilities in the graph database landscape. Recently, Neo4j Inc., Amazon, and others have started to investigate how to use GNNs within their systems [105]. Here, Neo4j already supports obtaining embeddings and using them for node or graph classification. However, there is still limited support for taking advantage of the full scope of information provided by LPG labels/properties. This has been alleviated by LPG2vec [37], an architecture for enabling Neural Graph Databases based on LPG. While in this work we do not focus specifically on GNNs, we illustrate that GDI can also be used to support graph ML workloads in the GDB domain, and thus it can serve as a propeller for the upcoming Neural Graph Database systems.

9 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, and the whole CSCS team granting access to the Ault and Daint machines and to the Slim Fly cluster, 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] [n.d.]. Apache TinkerGraph. Available at http://tinkerpop.apache.org.
[2] 2013. *foMPI.* http://spcl.inf.ethz.ch/Research/Parallel_Programming/foMPI.
[3] Yuichiro Ajima, Takahiro Kawashima, Takayuki Okamoto, Naoyuki Shida, Kouichi Hirai, Toshiyuki Shimizu, Shinya Hiramoto, Yoshiro Ikeda, Takahide Yoshikawa, Kenji Uchida, et al. 2018. The tofu interconnect d. In *2018 IEEE International Conference on Cluster Computing (CLUSTER).* IEEE, 646–654.
[4] Farahnaz Akrami, Mohammed Samiul Saeef, Qingheng Zhang, Wei Hu, and Chengkai Li. 2020. Realistic re-evaluation of knowledge graph completion methods: An experimental study. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data.* 1995–2010.
[5] Alibaba Cloud. 2018. Super computing cluster. https://www.alibabacloud.com/product/scc.
[6] Amazon. 2018. Amazon Neptune. Available at https://aws.amazon.com/neptune/.
[7] Renzo Angles, Marcelo Arenas, Pablo Barceló, Peter Boncz, George Fletcher, Claudio Gutierrez, Tobias Lindaaker, Marcus Paradies, Stefan Plantikow, Juan Sequeda, et al. 2018. G-CORE: A core for future graph query languages. In *International Conference on Management of Data.* 1421–1432.
[8] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. 2017. Foundations of Modern Query Languages for Graph Databases. *in ACM Comput. Surv.* 50, 5, Article 68 (2017), 40 pages. https://doi.org/10.1145/3104031
[9] 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. *ACM SIGMOD Record* 43, 1 (2014), 27–31.
[10] Renzo Angles and Claudio Gutierrez. 2008. Survey of Graph Database Models. *in ACM Comput. Surv.* 40, 1, Article 1 (2008), 39 pages. https://doi.org/10.1145/1322432.1322433
[11] Renzo Angles and Claudio Gutierrez. 2018. An Introduction to Graph Data Management. In *Graph Data Management, Fundamental Issues and Recent Developments.* 1–32.
[12] Apache. 2018. Apache Mormotta. Available at http://marmotta.apache.org/.
[13] ArangoDB Inc. 2018. ArangoDB. Available at https://docs.arangodb.com/3.3/Manual/DataModeling/Concepts.html.
[14] ArangoDB Inc. 2018. ArangoDB: Index Free Adjacency or Hybrid Indexes for Graph Databases. Available at https://www.arangodb.com/2016/04/index-free-adjacency-hybrid-indexes-graph-databases/.
[15] ArangoDB Inc. 2018. ArangoDB Starter Tool. Available at https://docs.arangodb.com/devel/Manual/Tutorials/Starter/.
[16] Baba Arimilli, Ravi Arimilli, Vicente Chung, Scott Clark, Wolfgang Denzel, Ben Drerup, Torsten Hoefler, Jody Joyner, Jerry Lewis, Jian Li, Nan Ni, and Ram Rajamony. 2010. The PERCS High-Performance Interconnect. In *Proceedings of the 2010 18th IEEE Symposium on High Performance Interconnects (HOTI '10).* IEEE Computer Society, Washington, DC, USA, 75–82. https://doi.org/10.1109/HOTI.2010.16
[17] Timothy G Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. 2013. LinkBench: a database benchmark based on the Facebook social graph. In *ACM SIGMOD.* ACM, 1185–1196.
[18] Siddhant Arora. 2020. A survey on graph neural networks for knowledge graph completion. *arXiv preprint arXiv:2007.12374* (2020).
[19] Ariful Azad, Oguz Selvitopi, Md Taufique Hussain, John R Gilbert, and Aydın Buluç. 2021. Combinatorial BLAS 2.0: Scaling combinatorial algorithms on distributed-memory systems. *IEEE Transactions on Parallel and Distributed Systems* 33, 4 (2021), 989–1001.
[20] Sumita Barahmand and Shahram Ghandeharizadeh. 2013. BG: A Benchmark to Evaluate Interactive Social Networking Actions. In *CIDR.*
[21] Claude Barthels, Simon Loesing, Gustavo Alonso, and Donald Kossmann. 2015. Rack-Scale In-Memory Join Processing Using RDMA. In *Proceedings of the 2015 International Conference on Management of Data (SIGMOD '15).* 1463–1475.
[22] Aaron Bernstein, Sebastian Forster, and Monika Henzinger. 2021. A deamortization approach for dynamic spanner and dynamic maximal matching. *ACM Transactions on Algorithms (TALG)* 17, 4 (2021), 1–51.
[23] Maciej Besta et al. 2019. Slim Graph: Practical Lossy Graph Compression for Approximate Graph Processing, Storage, and Analytics. , Article 35 (2019), 25 pages. https://doi.org/10.1145/3295500.3356182
[24] Maciej Besta et al. 2020. High-Performance Parallel Graph Coloring with Strong Guarantees on Work, Depth, and Quality. In *ACM/IEEE SC* (Atlanta, Georgia). Article 99, 17 pages.
[25] Maciej Besta et al. 2021. GraphMineSuite: Enabling High-Performance and Programmable Graph Mining Algorithms with Set Algebra. *VLDB.*
[26] Maciej Besta et al. 2021. SISA: Set-Centric Instruction Set Architecture for Graph Mining on Processing-in-Memory Systems. *arXiv preprint arXiv:2104.07582* (2021).
[27] Maciej Besta et al. 2022. Practice of Streaming Processing of Dynamic Graphs: Concepts, Models, and Systems. *IEEE TPDS* (2022).
[28] Maciej Besta, Marc Fischer, Tal Ben-Nun, Dimitri Stanojevic, Johannes De Fine Licht, and Torsten Hoefler. 2020. Substream-Centric Maximum Matchings on FPGA. *ACM TRETS* 13, 2 (2020), 1–33.
[29] Maciej Besta, Robert Gerstenberger, Nils Blach, Marc Fischer, and Torsten Hoefler. 2023. *GDI: A Graph Database Interface Standard.* Technical Report. Available at https://spcl.inf.ethz.ch/Research/Parallel_Programming/GDI/.
[30] Maciej Besta, Raphael Grob, Cesare Miglioli, Nicola Bernold, Grzegorz Kwasniewski, Gabriel Gjini, Raghavendra Kanakagiri, Saleh Ashkboos, Lukas Gianinazzi, Nikoli Dryden, et al. 2022. Motif Prediction with Graph Neural Networks, In ACM KDD. *arXiv preprint arXiv:2106.00761.*
[31] Maciej Besta and Torsten Hoefler. 2014. Fault tolerance for remote memory access programming models. In *ACM HPDC* (Vancouver, BC, Canada). 37–48. https://doi.org/10.1145/2600212.2600224
[32] Maciej Besta and Torsten Hoefler. 2015. Accelerating irregular computations with hardware transactional memory and active messages. In *ACM HPDC* (Portland, Oregon, USA). 161–172. https://doi.org/10.1145/2749246.2749263
[33] Maciej Besta and Torsten Hoefler. 2015. Accelerating Irregular Computations with Hardware Transactional Memory and Active Messages. In *Proc. of the Intl. Symp. on High-Perf. Par. and Dist. Comp. (HPDC '15).* 161–172.
[34] Maciej Besta and Torsten Hoefler. 2015. Active access: A mechanism for high-performance distributed data-centric computations. In *ACM ICS* (Newport Beach, California, USA). 155–164. https://doi.org/10.1145/2751205.2751219
[35] Maciej Besta and Torsten Hoefler. 2018. Survey and taxonomy of lossless graph compression and space-efficient graph representations. *arXiv preprint arXiv:1806.01799* (2018).
[36] Maciej Besta and Torsten Hoefler. 2022. Parallel and Distributed Graph Neural Networks: An In-Depth Concurrency Analysis. *arXiv preprint arXiv:2205.09702* (2022).
[37] Maciej Besta, Patrick Iff, Florian Scheidl, Kazuki Osawa, Nikoli Dryden, Michal Podstawski, Tiancheng Chen, and Torsten Hoefler. 2022. Neural Graph Databases. In *LOG.*
[38] Maciej Besta, Raghavendra Kanakagiri, Harun Mustafa, Mikhail Karasikov, Gunnar Rätsch, Torsten Hoefler, and Edgar Solomonik. 2019. Communication-Efficient Jaccard Similarity for High-Performance Distributed Genome Comparisons. *arXiv preprint arXiv:1911.04200* (2019).
[39] Maciej Besta, Florian Marending, Edgar Solomonik, and Torsten Hoefler. 2017. Slimsell: A vectorizable graph representation for breadth-first search. In *IEEE IPDPS.* IEEE, 32–41.
[40] Maciej Besta, Cesare Miglioli, Paolo Sylos Labini, Jakub Tětek, Patrick Iff, Raghavendra Kanakagiri, Saleh Ashkboos, Kacper Janda, Michal Podstawski, Grzegorz Kwasniewski, et al. 2022. ProbGraph: High-Performance and High-Accuracy Graph Mining with Probabilistic Set Representations. In *ACM/IEEE Supercomputing.*
[41] Maciej Besta, Emanuel Peter, Robert Gerstenberger, Marc Fischer, Michał Podstawski, Claude Barthels, Gustavo Alonso, and Torsten Hoefler. 2019. Demystifying Graph Databases: Analysis and Taxonomy of Data Organization, System Designs, and Graph Queries. *arXiv preprint arXiv:1910.09017* (2019).
[42] 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 *ACM HPDC.*
[43] Maciej Besta, Dimitri Stanojevic, Tijana Zivic, Jagpreet Singh, Maurice Hoerold, and Torsten Hoefler. 2018. Log (graph): a near-optimal high-performance graph representation.. In *PACT* (Limassol, Cyprus). ACM, Article 7, 13 pages. https://doi.org/10.1145/3243176.3243198
[44] Gianfranco Bilardi and Andrea Pietracaprina. 2011. *Models of Computation, Theoretical.* Springer US, Boston, MA, 1150–1158. https://doi.org/10.1007/978-0-387-09766-4_218
[45] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. 2016. The End of Slow Networks: It's Time for a Redesign. *Proceedings of*

the VLDB Endowment 9, 7 (March 2016), 528–539.
[46] Blazegraph. 2018. BlazeGraph DB. Available at https://www.blazegraph.com/.
[47] Guy E. Blelloch and Bruce M. Maggs. 2010. *Parallel Algorithms*. Chapman & Hall/CRC.
[48] Paolo Boldi and Sebastiano Vigna. 2004. The webgraph framework I: compression techniques. In *ACM WWW*. ACM, 595–602.
[49] Angela Bonifati, George Fletcher, Hannes Voigt, and Nikolay Yakovets. 2018. Querying graphs. *Synthesis Lectures on Data Management* 10, 3 (2018), 1–184.
[50] Benjamin Brock, Aydin Buluç, Timothy G. Mattson, Scott McMillan, José E. Moreira, Roger Pearce, Oguz Selvitopi, and Trevor Steil. 2020. Considerations for a Distributed GraphBLAS API. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2020, New Orleans, LA, USA, May 18-22, 2020*. IEEE, 215–218. https://doi.org/10.1109/IPDPSW50202.2020.00048
[51] 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.
[52] Aydın Buluç and John R Gilbert. 2011. The Combinatorial BLAS: Design, implementation, and applications. *IJHPCA* 25, 4 (2011), 496–509.
[53] Chiranjeeb Buragohain, Knut Magne Risvik, Paul Brett, Miguel Castro, Wonhee Cho, Joshua Cowhig, Nikolas Gloy, Karthik Kalyanaraman, Richendra Khanna, John Pao, et al. 2020. A1: A Distributed In-Memory Graph Database. In *Proceedings of the 2020 International Conference on Management of Data (SIGMOD '20)*. 329–344.
[54] Matthew Burke, Sowmya Dharanipragada, Shannon Joyner, Adriana Szekeres, Jacob Nelson, Irene Zhang, and Dan R. K. Ports. 2021. PRISM: Rethinking the RDMA Interface for Distributed Systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)*. 228–242.
[55] Callidus Software Inc. 2018. OrientDB: Lightweight Edges. Available at https://orientdb.com/docs/3.0.x/java/Lightweight-Edges.html.
[56] Cambridge Semantics. 2018. AnzoGraph. Available at https://www.cambridgesemantics.com/product/anzograph/.
[57] Huanqi Cao, Yuanwei Wang, Haojie Wang, Heng Lin, Zixuan Ma, Wanwang Yin, and Wenguang Chen. 2022. Scaling graph traversal to 281 trillion edges with 40 million cores. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 234–245.
[58] Mihai Capotă, Tim Hegeman, Alexandru Iosup, Arnau Prat-Pérez, Orri Erling, and Peter Boncz. 2015. Graphalytics: A big data benchmark for graph-processing platforms. In *Proceedings of the GRADES'15*. 1–6. https://doi.org/10.1145/2764947.2764954
[59] Cayley. 2018. CayleyGraph. Available at https://cayley.io/ and https://github.com/cayleygraph/cayley.
[60] Deepayan Chakrabarti and Christos Faloutsos. 2006. Graph mining: Laws, generators, and algorithms. *ACM computing surveys (CSUR)* 38, 1 (2006), 2.
[61] Brad Chamberlain, Steve Deitz, Mary Beth Hribar, and Wayne Wong. 2005. *Chapel*. Technical Report. Cray Inc.
[62] Ines Chami, Sami Abu-El-Haija, Bryan Perozzi, Christopher Ré, and Kevin Murphy. 2020. Machine learning on graphs: A model and comprehensive taxonomy. *arXiv preprint arXiv:2005.03675* (2020).
[63] Ernie Chan, Marcel Heimlich, Avi Purkayastha, and Robert Van De Geijn. 2007. Collective communication: theory, practice, and experience. *Concurrency and Computation: Practice and Experience* (2007).
[64] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. X10: an Object-Oriented Approach to Non-Uniform Cluster Computing. *SIGPLAN Not.* 40, 10 (Oct. 2005), 519–538. https://doi.org/10.1145/1103845.1094852
[65] Hongzhi Chen, Changji Li, Juncheng Fang, Chenghuan Huang, James Cheng, Jian Zhang, Yifan Hou, and Xiao Yan. 2019. Grasper: A high performance distributed system for OLAP on property graphs. In *Proceedings of the ACM Symposium on Cloud Computing*. 87–100.
[66] Hongzhi Chen, Changji Li, Chenguang Zheng, Chenghuan Huang, Juncheng Fang, James Cheng, and Jian Zhang. 2021. G-Tran: Making Distributed Graph Transactions Fast. *arXiv preprint arXiv:2105.04449* (2021).
[67] Zhe Chen, Yuehan Wang, Bin Zhao, Jing Cheng, Xin Zhao, and Zongtao Duan. 2020. Knowledge graph completion: A review. *Ieee Access* 8 (2020), 192435–192456.
[68] Sutanay Choudhury, Khushbu Agarwal, Sumit Purohit, Baichuan Zhang, Meg Pirrung, Will Smith, and Mathew Thomas. 2017. Nous: Construction and querying of dynamic knowledge graphs. In *IEEE ICDE*. 1563–1565.
[69] Marek Ciglan, Alex Averbuch, and Ladialav Hluchy. 2012. Benchmarking traversal operations over graph databases. In *2012 IEEE 28th International Conference on Data Engineering Workshops*. IEEE, 186–189.
[70] Diane J Cook and Lawrence B Holder. 2006. *Mining graph data*. John Wiley & Sons.
[71] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. 2020. SeBS: A Serverless Benchmark Suite for Function-as-a-Service Computing. *arXiv preprint arXiv:2012.14132* (2020).
[72] Yuanfei Dai, Shiping Wang, Neal N Xiong, and Wenzhong Guo. 2020. A survey on knowledge graph embedding: Approaches, applications and benchmarks. *Electronics* 9, 5 (2020), 750.
[73] DataStax, Inc. 2018. DSE Graph (DataStax). Available at https://www.datastax.com/.
[74] Alex Davies, Petar Veličković, Lars Buesing, Sam Blackwell, Daniel Zheng, Nenad Tomašev, Richard Tanburn, Peter Battaglia, Charles Blundell, András Juhász, et al. 2021. Advancing mathematics by guiding human intuition with AI. *Nature* 600, 7887 (2021), 70–74.
[75] Ali Davoudian, Liu Chen, and Mengchi Liu. 2018. A survey on NoSQL stores. *ACM Computing Surveys (CSUR)* 51, 2, Article 40 (2018), 43 pages. https://doi.org/10.1145/3158661
[76] Camil Demetrescu, David Eppstein, Zvi Galil, and Giuseppe F Italiano. 2009. Dynamic graph algorithms. In *Algorithms and Theory of Computation Handbook, Volume 1*. Chapman and Hall/CRC, 235–262.
[77] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor E Lee. 2020. Aggregation support for modern graph analytics in TigerGraph. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 377–392.
[78] Dgraph Labs, Inc. 2018. DGraph. Available at https://dgraph.io/, https://docs.dgraph.io/design-concepts.
[79] David Dominguez-Sal, Peter Urbón-Bayes, Aleix Giménez-Vanó, Sergio Gómez-Villamor, Norbert Martinez-Bazan, and Josep-Lluis Larriba-Pey. 2010. Survey of graph database performance on the HPC scalable graph analysis benchmark. In *International Conference on Web-Age Information Management*. Springer, 37–48.
[80] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. 401–414.
[81] Ran Duan, Haoqing He, and Tianyi Zhang. 2019. Dynamic edge coloring with improved approximation. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 1937–1945.
[82] Ayush Dubey, Greg D Hill, Robert Escriva, and Emin Gün Sirer. 2016. Weaver: a high-performance, transactional graph database based on refinable timestamps. *Proceedings of the VLDB Endowment* 9, 11 (2016), 852–863.
[83] David Ediger, Rob McColl, Jason Riedy, and David A Bader. 2012. Stinger: High performance data structure for streaming graphs. In *IEEE HPEC*. IEEE, 1–5.
[84] David Eppstein, Zvi Galil, and Giuseppe F Italiano. 1999. Dynamic graph algorithms. *Algorithms and theory of computation handbook* 1 (1999), 9–1.
[85] 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 *ACM SIGMOD (in SIGMOD)*. 619–630.
[86] Greg Faanes, Abdulla Bataineh, 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 *SC*.
[87] FactNexus. 2018. GraphBase. Available at https://graphbase.ai/.
[88] Guanyu Feng, Zixuan Ma, Daixuan Li, Shengqi Chen, Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2021. RisGraph: A Real-Time Streaming System for Evolving Graphs to Support Sub-millisecond Per-update Analysis at Millions Ops/s, In Proceedings of the 2021 International Conference on Management of Data. *arXiv preprint arXiv:2004.00803*, 513–527.
[89] Franz Inc. 2018. AllegroGraph. Available at https://franz.com/agraph/allegrograph/.
[90] Santhosh Kumar Gajendran. 2012. A survey on NoSQL databases. *University of Illinois* (2012).
[91] R. Gerstenberger et al. 2013. Enabling Highly-scalable Remote Memory Access Programming with MPI-3 One Sided. In *ACM/IEEE Supercomputing* (Denver, Colorado) *(SC '13)*. Article 53, 12 pages.
[92] Robert Gerstenberger, Maciej Besta, and Torsten Hoefler. 2013. Enabling highly-scalable remote memory access programming with mpi-3 one sided. In *ACM/IEEE Supercomputing*. 1–12.
[93] Lukas Gianinazzi, Maciej Besta, Yannick Schaffner, and Torsten Hoefler. 2021. Parallel Algorithms for Finding Large Cliques in Sparse Graphs. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*. 243–253.
[94] Lukas Gianinazzi, Maximilian Fries, Nikoli Dryden, Tal Ben-Nun, and Torsten Hoefler. 2021. Learning Combinatorial Node Labeling Algorithms. *arXiv preprint arXiv:2106.03594* (2021).
[95] Lukas Gianinazzi, Pavel Kalvoda, Alessandro De Palma, Maciej Besta, and Torsten Hoefler. 2018. Communication-avoiding parallel minimum cuts and connected components. In *ACM SIGPLAN Notices*.
[96] R.E. Grant, M.J. Rashti, A. Afsahi, and P. Balaji. 2011. RDMA Capable iWARP over Datagrams. In *Par. Dist. Proc. Symp. (IPDPS), 2011 IEEE Intl.* 628–639. https://doi.org/10.1109/IPDPS.2011.66
[97] 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. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2242–2254. https://doi.org/10.14778/3352063.3352139
[98] William L Hamilton. 2020. Graph representation learning. *Synthesis Lectures on Artifical Intelligence and Machine Learning* 14, 3 (2020), 1–159.
[99] William L Hamilton et al. 2017. Representation learning on graphs: Methods and applications. *arXiv preprint arXiv:1709.05584* (2017).

[100] Jing Han, E Haihong, Guan Le, and Jian Du. 2011. Survey on NoSQL database. In *2011 6th international conference on pervasive computing and applications*. IEEE, 363–366.

[101] Steve Harris, Nick Lamb, Nigel Shadbolt, et al. 2009. 4store: The design and implementation of a clustered RDF store. In *5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2009)*, Vol. 94.

[102] Monika R Henzinger and Valerie King. 1999. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *Journal of the ACM (JACM)* 46, 4 (1999), 502–516.

[103] M. Herlihy and N. Shavit. 2008. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc.

[104] Paul N Hilfinger, Dan Oscar Bonachea, Kaushik Datta, David Gay, Susan L Graham, Benjamin Robert Liblit, Geoffrey Pike, Jimmy Zhigang Su, and Katherine A Yelick. 2005. *Titanium Language Reference Manual, version 2.19*. Technical Report. UC Berkeley Tech Rep. UCB/EECS-2005-15.

[105] Amy E Hodler and Mark Needham. 2022. Graph Data Science Using Neo4j. In *Massive Graph Analytics*. Chapman and Hall/CRC, 433–457.

[106] 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*. 1–12.

[107] Torsten Hoefler and Dmitry Moor. 2014. Energy, memory, and runtime tradeoffs for implementing collective communication operations. *Supercomputing frontiers and innovations* (2014).

[108] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. 2008. Accurately measuring collective operations at massive scale. In *2008 IEEE International Symposium on Parallel and Distributed Processing*. IEEE, 1–8.

[109] Tamás Horváth, Thomas Gärtner, and Stefan Wrobel. 2004. Cyclic pattern kernels for predictive graph mining. In *KDD*. ACM, 158–167.

[110] Jian Huang, Xiangyong Ouyang, Jithin Jose, Md Wasi-ur Rahman, Hao Wang, Miao Luo, Hari Subramoni, Chet Murthy, and Dhabaleswar K Panda. 2012. High-performance design of hbase with rdma over infiniband. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. IEEE, 774–785.

[111] InfiniBand Trade Association. 2010. *Supplement to InfiniBand Architecture Spec., Vol. 1, Rel. 1.2.1. Annex A16: RDMA over Converged Ethernet (RoCE)*.

[112] Alexandru Iosup, Tim Hegeman, Wing Lung Ngai, Stijn Heldens, Arnau Prat-Pérez, Thomas Manhardto, Hassan Chafio, Mihai Capotă, Narayanan Sundaram, Michael Anderson, Ilie Gabriel Tănase, Yinglong Xia, Lifeng Nai, and Peter Boncz. 2016. LDBC Graphalytics: A Benchmark for Large-scale Graph Analysis on Parallel and Distributed Platforms. *in VLDB* 9, 13 (2016), 1317–1328. https://doi.org/10.14778/3007263.3007270

[113] Nusrat S Islam, Mohammad Wahidur Rahman, Jithin Jose, Raghunath Rajachandrasekar, Hao Wang, Hari Subramoni, Chet Murthy, and Dhabaleswar K Panda. 2012. High performance RDMA-based design of HDFS over InfiniBand. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–12.

[114] ISO Fortran Committee. 2010. Fortran 2008 Standard (ISO/IEC 1539-1:2010). (2010).

[115] Zoran Ivković and Errol L Lloyd. 1993. Fully dynamic maintenance of vertex cover. In *International Workshop on Graph-Theoretic Concepts in Computer Science*. Springer, 99–111.

[116] Sagar Jha, Jonathan Behrens, Theo Gkountouvas, Matthew Milano, Weijia Song, Edward Tremel, Robbert Van Renesse, Sydney Zink, and Kenneth P. Birman. 2019. Derecho: Fast State Machine Replication for Cloud Services. *ACM Transactions on Computer Systems* 36, 2, Article 4 (April 2019), 49 pages.

[117] Chuntao Jiang, Frans Coenen, and Michele Zito. 2013. A survey of frequent subgraph mining algorithms. *The Knowledge Engineering Review* 28, 1 (2013), 75–105.

[118] Salim Jouili and Valentin Vansteenberghe. 2013. An empirical comparison of graph databases. In *2013 International Conference on Social Computing*. IEEE, 708–715.

[119] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, et al. 2021. Highly accurate protein structure prediction with AlphaFold. *Nature* 596, 7873 (2021), 583–589.

[120] Martin Junghanns, 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*. Springer, 457–505.

[121] Matthias Jurisch and Bodo Igler. 2018. RDF2Vec-based classification of ontology alignment changes. *arXiv preprint arXiv:1805.09145* (2018).

[122] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2014. Using RDMA efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM*. 295–306.

[123] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2016. Design guidelines for high performance {RDMA} systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. 437–450.

[124] R. Kumar Kaliyar. 2015. Graph databases: A survey. In *ICCCA*. 785–790.

[125] U. Kang, Hanghang Tong, Jimeng Sun, Ching-Yung Lin, and Christos Faloutsos. 2012. Gbase: An Efficient Analysis Platform for Large Graphs. *In PVLDB* 21, 5 (2012), 637–650. https://doi.org/10.1007/s00778-012-0283-9

[126] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedbhi, Jeremy Chen, et al. 2017. Graphflow: An Active Graph Database. In *ACM SIGMOD* (Chicago, Illinois, USA). 1695–1698. https://doi.org/10.1145/3035918.3056445

[127] Tejas Karmarkar. 2015. Availability of linux rdma on microsoft azure. *Microsoft Azure Blog* (2015).

[128] Jeremy Kepner, Peter Aaltonen, David Bader, Aydin Buluç, Franz Franchetti, John Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, et al. 2016. Mathematical foundations of the GraphBLAS. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–9.

[129] Jeremy Kepner, David Bader, Aydın Buluç, John Gilbert, Timothy Mattson, and Henning Meyerhenke. 2015. Graphs, matrices, and the GraphBLAS: Seven good reasons. *Procedia Computer Science* 51 (2015), 2453–2462.

[130] Daehyeok Kim, Amirsaman Memaripour, Anirudh Badam, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Shachar Raindel, Steven Swanson, Vyas Sekar, and Srinivasan Seshan. 2018. Hyperloop: Group-Based NIC-Offloading to Accelerate Replicated Transactions in Multi-Tenant Storage Systems. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*. 297–312.

[131] John Kim, Wiliam J. Dally, Steve Scott, and Dennis Abts. 2008. Technology-Driven, Highly-Scalable Dragonfly Topology, In Proc. of Intl. Symp. Comp. Arch. *ISCA*, 77–88. https://doi.org/10.1109/ISCA.2008.19

[132] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

[133] Vijay Kumar and Anjan Babu. 2015. Domain Suitable Graph Database Selection: A Preliminary Report. In *3rd International Conference on Advances in Engineering Sciences & Applied Mathematics, London, UK*. 26–29.

[134] Jérôme Kunegis. 2013. Konect: the koblenz network collection. In *Proc. of Intl. Conf. on World Wide Web (WWW)*. ACM, 1343–1350.

[135] Ora Lassila, Ralph R Swick, et al. 1998. Resource description framework (RDF) model and syntax specification. (1998).

[136] Min-Joong Lee, Sunghee Choi, and Chin-Wan Chung. 2016. Efficient algorithms for updating betweenness centrality in fully dynamic graphs. *Information Sciences* 326 (2016), 278–296.

[137] 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, Feb (2010), 985–1042.

[138] Changji Li, Hongzhi Chen, Shuai Zhang, Yingqian Hu, Chao Chen, Zhenjie Zhang, Meng Li, Xiangchen Li, Dongqing Han, Xiaohui Chen, et al. 2022. ByteGraph: A High-Performance Distributed Graph Database in ByteDance. *VLDB* (2022).

[139] Feng Li, Sudipto Das, Manoj Syamala, and Vivek R. Narasayya. 2016. Accelerating Relational Databases by Leveraging Remote Memory and RDMA. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. 355–370.

[140] Heng Lin, Xiaowei Zhu, Bowen Yu, Xiongchao Tang, Wei Xue, Wenguang Chen, Lufei Zhang, Torsten Hoefler, Xiaosong Ma, Xin Liu, et al. 2018. ShenTu: processing multi-trillion edge graphs on millions of cores in seconds. In *ACM/IEEE Supercomputing* (Dallas, Texas). IEEE Press, Article 56, 11 pages.

[141] Yankai Lin, Zhiyuan Liu, Maosong Sun, Yang Liu, and Xuan Zhu. 2015. Learning entity and relation embeddings for knowledge graph completion. In *AAAI*.

[142] Matteo Lissandrini, Martin Brugnara, and Yannis Velegrakis. 2017. *An Evaluation Methodology and Experimental Comparison of Graph Databases*. Technical Report. Technical report, University of Trento.

[143] Matteo Lissandrini, Martin Brugnara, and Yannis Velegrakis. 2018. Beyond macrobenchmarks: microbenchmark-based graph database evaluation. *Proceedings of the VLDB Endowment* 12, 4 (2018), 390–403. https://doi.org/10.14778/3297753.3297759

[144] Jiuxing Liu, Jiesheng Wu, and Dhabaleswar K Panda. 2004. High performance RDMA-based MPI implementation over InfiniBand. *International Journal of Parallel Programming* 32, 3 (2004), 167–198.

[145] Xiaoyi Lu, Nusrat S Islam, Md Wasi-Ur-Rahman, Jithin Jose, Hari Subramoni, Hao Wang, and Dhabaleswar K Panda. 2013. High-performance design of Hadoop RPC with RDMA over InfiniBand. In *2013 42nd International Conference on Parallel Processing*. IEEE, 641–650.

[146] Bingqing Lyu, Lu Qin, Xuemin Lin, Lijun Chang, and Jeffrey Xu Yu. 2016. Scalable supergraph search in large graph databases. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. IEEE, 157–168.

[147] Shuai Ma, Jia Li, Chunming Hu, Xuelian Lin, and Jinpeng Huai. 2016. Big graph search: challenges and techniques. *Frontiers of Computer Science* 10, 3 (2016), 387–398.

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

[149] Kamesh Madduri, David Ediger, Karl Jiang, David A Bader, and Daniel Chavarria-Miranda. 2009. A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE, 1–8.

[150] 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* (2022).

[151] Norbert Martínez-Bazan, Victor Muntés-Mulero, Sergio Gómez-Villamor, M.Ángel Águila Lorente, David Dominguez-Sal, and Josep-L. Larriba-Pey. 2012. Efficient Graph Management Based On Bitmap Indices. *In IDEAS* (2012), 110–119. https://doi.org/10.1145/2351476.2351489

[152] Tim Mattson et al. 2014. Standards for graph algorithm primitives. *arXiv preprint arXiv:1408.0393* (2014).

[153] Robert Campbell McColl, David Ediger, Jason Poovey, Dan Campbell, and David A Bader. 2014. A performance evaluation of open source graph databases. In *Proceedings of the first workshop on Parallel programming for analytics applications* (Orlando, Florida, USA). ACM, 11–18. https://doi.org/10.1145/2567634.2567638

[154] Memgraph Ltd. 2018. Memgraph. Available at https://memgraph.com/.

[155] Microsoft. 2018. Azure Cosmos DB. Available at https://azure.microsoft.com/en-us/services/cosmos-db/.

[156] Justin J Miller. 2013. Graph Database Applications and Concepts with Neo4j. In *Proceedings of the Southern Association for Information Systems Conference*, Vol. 2324.

[157] Azalia Mirhoseini, Anna Goldie, Mustafa Yazgan, Joe Wenjie Jiang, Ebrahim Songhori, Shen Wang, Young-Joon Lee, Eric Johnson, Omkar Pathak, Azade Nazi, et al. 2021. A graph placement methodology for fast chip design. *Nature* 594, 7862 (2021), 207–212.

[158] Christopher Mitchell, Yifeng Geng, and Jinyang Li. 2013. Using {One-Sided}{RDMA} Reads to Build a Fast,{CPU-Efficient}{Key-Value} Store. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. 103–114.

[159] Gianfranco E Modoni, Marco Sacco, and Walter Terkaj. 2014. A survey of RDF store solutions. In *2014 International Conference on Engineering, Technology and Innovation (ICE)*. IEEE, 1–7.

[160] MPI Forum. 2012. MPI: A Message-Passing Interface Standard. Version 3. available at: http://www.mpi-forum.org (Sep. 2012).

[161] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. 2010. Introducing the graph 500. *Cray Users Group (CUG)* 19 (2010), 45–74.

[162] Chemseddine Nabti and Hamida Seba. 2017. Querying massive graph data: A compress and search approach. *Future Generation Computer Systems* 74 (2017), 63–75.

[163] Ofer Neiman and Shay Solomon. 2015. Simple deterministic algorithms for fully dynamic maximal matching. *ACM Transactions on Algorithms (TALG)* 12, 1 (2015), 1–15.

[164] Networked Planet Limited. 2018. BrightstarDB. Available at http://brightstardb.com/.

[165] Objectivity Inc. 2018. InfiniteGraph. Available at https://www.objectivity.com/products/infinitegraph/.

[166] Ontotext. 2018. GraphDB. Available at https://www.ontotext.com/products/graphdb/.

[167] OpenLink. 2018. Virtuoso. Available at https://virtuoso.openlinksw.com/.

[168] Oracle. 2018. Oracle Spatial and Graph. Available at https://www.oracle.com/database/technologies/spatialandgraph.html.

[169] Oracle. 2020. Oracle Cloud. https://www.oracle.com/cloud/hpc/.

[170] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank citation ranking: Bringing order to the web.* Technical Report. Stanford InfoLab.

[171] 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*. 397–400.

[172] 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.

[173] Tobias Pfaff, Meire Fortunato, Alvaro Sanchez-Gonzalez, and Peter W Battaglia. 2020. Learning mesh-based simulation with graph networks. *arXiv preprint arXiv:2010.03409* (2020).

[174] Marius Poke and Torsten Hoefler. 2015. Dare: High-performance state machine replication on rdma networks. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*. 107–118.

[175] Jaroslav Pokorny. 2015. Graph databases: their power and limitations. In *IFIP International Conference on Computer Information Systems and Industrial Management*. Springer, 58–69.

[176] Matteo Pontecorvi and Vijaya Ramachandran. 2015. A faster algorithm for fully dynamic betweenness centrality. *arXiv preprint arXiv:1506.05783* (2015).

[177] Jan Portisch, Michael Hladik, and Heiko Paulheim. 2020. RDF2Vec Light–A Lightweight Approach for Knowledge Graph Embeddings. *arXiv preprint arXiv:2009.07659* (2020).

[178] Profium. 2018. Profium Sense. Available at https://www.profium.com/en/.

[179] Zhengping Qian, Chenqiang Min, Longbin Lai, Yong Fang, Gaofeng Li, Youyang Yao, Bingqing Lyu, Xiaoli Zhou, Zhimin Chen, and Jingren Zhou. 2021. GAIA: A System for Interactive Analysis on Distributed Graphs Using a High-Level Language. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 321–335.

[180] Hongchao Qin, Rong-Hua Li, Guoren Wang, Lu Qin, Yurong Cheng, and Ye Yuan. 2019. Mining Periodic Cliques in Temporal Networks. In *ICDE*. IEEE, 1130–1141.

[181] R Recio, B Metzler, P Culley, J Hilland, and D Garcia. 2007. *A remote direct memory access protocol specification.* Technical Report. RFC 5040, October. RFC 5040.

[182] Redis Labs. 2018. RedisGraph. Available at https://oss.redislabs.com/redisgraph/.

[183] 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 *CUG*. Cray Users Group.

[184] Petar Ristoski and Heiko Paulheim. 2016. Rdf2vec: Rdf graph embeddings for data mining. In *International Semantic Web Conference*. Springer, 498–514.

[185] Petar Ristoski, Jessica Rosati, Tommaso Di Noia, Renato De Leone, and Heiko Paulheim. 2019. RDF2Vec: RDF graph embeddings and their applications. *Semantic Web* 10, 4 (2019), 721–752.

[186] Ian Robinson, Jim Webber, and Emil Eifrem. 2015. Graph Database Internals. In *Graph Databases, Second Edition*. O'Relly, Chapter 7, 149–170.

[187] Wolf Rödiger, Tobias Mühlbauer, Alfons Kemper, and Thomas Neumann. 2015. High-Speed Query Processing over High-Speed Networks. *Proceedings of the VLDB Endowment* 9, 4 (December 2015), 228–239.

[188] Florin Rusu and Zhiyi Huang. 2019. In-Depth Benchmarking of Graph Database Systems with the Linked Data Benchmark Council (LDBC) Social Network Benchmark (SNB). *arXiv preprint arXiv:1907.07405* (2019).

[189] Sherif Sakr et al. 2020. The Future is Big Graphs! A Community View on Graph Processing Systems. *arXiv preprint arXiv:2012.06171* (2020).

[190] Alvaro Sanchez-Gonzalez, Jonathan Godwin, Tobias Pfaff, Rex Ying, Jure Leskovec, and Peter Battaglia. 2020. Learning to simulate complex physics with graph networks. In *ICML*.

[191] 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.

[192] Patrick Schmid, Maciej Besta, and Torsten Hoefler. 2016. High-performance distributed rma locks. In *ACM HPDC* (Kyoto, Japan). 19–30. https://doi.org/10.1145/2907294.2907323

[193] Hermann Schweizer, Maciej Besta, and Torsten Hoefler. 2015. Evaluating the cost of atomic operations on modern architectures. In *IEEE PACT*. IEEE, 445–456.

[194] Dipanjan Sengupta and Shuaiwen Leon Song. 2017. Evograph: On-the-fly efficient mining of evolving graphs on gpu. In *International Supercomputing Conference*. Springer, 97–119.

[195] Dipanjan Sengupta, Narayanan Sundaram, Xia Zhu, Theodore L Willke, Jeffrey Young, Matthew Wolf, and Karsten Schwan. 2016. Graphin: An online high performance incremental graph processing framework. In *European Conference on Parallel Processing*. Springer, 319–333.

[196] Baoxu Shi and Tim Weninger. 2017. Proje: Embedding projection for knowledge graph completion. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 31.

[197] Baoxu Shi and Tim Weninger. 2018. Open-world knowledge graph completion. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 32.

[198] Jiaxin Shi, Youyang Yao, Rong Chen, Haibo Chen, and Feifei Li. 2016. Fast and Concurrent RDF Queries with RDMA-Based Distributed Graph Exploration. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 317–332.

[199] Anna Kornfeld Simpson, Adriana Szekeres, Jacob Nelson, and Irene Zhang. 2020. Securing {RDMA} for {High-Performance} Datacenter Storage Systems. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*.

[200] Marc Snir, William Gropp, Steve Otto, Steven Huss-Lederman, Jack Dongarra, and David Walker. 1998. *MPI–the Complete Reference: the MPI core*. Vol. 1.

[201] Edgar Solomonik, Maciej Besta, Flavio Vella, and Torsten Hoefler. 2017. Scaling betweenness centrality using communication-efficient sparse matrix multiplication. In *ACM/IEEE Supercomputing*.

[202] Stardog Union. 2018. Stardog. Available at https://www.stardog.com/.

[203] Zhiqing Sun, Shikhar Vashishth, Soumya Sanyal, Partha Talukdar, and Yiming Yang. 2019. A re-evaluation of knowledge graph completion methods. *arXiv preprint arXiv:1911.03903* (2019).

[204] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R Dulloor, Michael J Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. 2015. Graphmat: High performance graph analytics made

productive. *Proceedings of the VLDB Endowment* 8, 11 (2015), 1214–1225. https://doi.org/10.14778/2809974.2809983
[205] 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 *ACM GRADES-NDA* (Houston, Texas) *(GRADES-NDA)*. Article 9, 11 pages. https://doi.org/10.1145/3210259.3210268
[206] Gábor Szárnyas, Jack Waudby, Benjamin A. Steer, Dávid Szakállas, Altan Birler, Mingxi Wu, Yuchen Zhang, and Peter A. Boncz. 2022. The LDBC Social Network Benchmark: Business Intelligence Workload. *VLDB* (2022).
[207] Yacine Taleb, Ryan Stutsman, Gabriel Antoniu, and Toni Cortes. 2018. Tailwind: Fast and Atomic RDMA-based Replication. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 851–863.
[208] Daniel ten Wolde et al. 2023. DuckPGQ: Efficient property graph queries in an analytical RDBMS. CIDR.
[209] Claudio Tesoriero. 2013. *Getting started with OrientDB*. Packt Publishing Birmingham, England.
[210] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. 2005. Optimization of collective communication operations in MPICH. *The International Journal of High Performance Computing Applications* 19, 1 (2005), 49–66.
[211] The Apache Software Foundation. 2021. Apache Jena TBD. Available at https://jena.apache.org/documentation/tdb/index.html.
[212] The InfiniBand Trade Association. 2004. *Infiniband Architecture Spec. Vol. 1, Rel. 1.2*. InfiniBand Trade Association.
[213] The Linux Foundation. 2017. JanusGraph - Distributed Graph Database.
[214] The Linux Foundation. 2018. JanusGraph. Available at http://janusgraph.org/.
[215] Mikkel Thorup. 2000. Near-optimal fully-dynamic graph connectivity. In *Proceedings of the thirty-second annual ACM symposium on Theory of computing*. 343–350.
[216] Yuanyuan Tian, Wen Sun, Sui Jun Tong, En Liang Xu, Mir Hamid Pirahesh, and Wei Zhao. 2019. Synergistic graph and SQL analytics inside IBM Db2. *Proceedings of the VLDB Endowment* 12, 12 (2019), 1782–1785.
[217] TigerGraph. 2018. TigerGraph. Available at https://www.tigergraph.com/.
[218] TigerGraph, Inc. 2022. *Using the Linked Data Benchmark Council Social Network Benchmark Methodology to Evaluate TigerGraph at 36 Terabytes*. White Paper.
[219] Lucian Toader, Alexandru Uta, Ahmed Musaafir, and Alexandru Iosup. 2019. Graphless: Toward serverless graph processing. In *2019 18th International Symposium on Parallel and Distributed Computing (ISPDC)*. IEEE, 66–73.
[220] Théo Trouillon, Christopher R Dance, Johannes Welbl, Sebastian Riedel, Éric Gaussier, and Guillaume Bouchard. 2017. Knowledge graph completion via complex tensor factorization. *arXiv preprint arXiv:1702.06879* (2017).
[221] Ioanna Tsalouchidou, Ricardo Baeza-Yates, Francesco Bonchi, Kewen Liao, and Timos Sellis. 2020. Temporal betweenness centrality in dynamic graphs. *International Journal of Data Science and Analytics* 9 (2020), 257–272.
[222] UPC Consortium. 2005. *UPC language specifications, v1.2*. Technical Report. Lawrence Berkeley National Laboratory. LBNL-59208.
[223] Jacopo Urbani, Sourav Dutta, Sairam Gurajada, and Gerhard Weikum. 2016. KOGNAC: efficient encoding of large knowledge graphs. *arXiv preprint arXiv:1604.04795* (2016).
[224] Quan Wang, Zhendong Mao, Bin Wang, and Li Guo. 2017. Knowledge graph embedding: A survey of approaches and applications. *IEEE TKDE* (2017).
[225] 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 *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–11.
[226] Web Data Commons. [n.d.]. Hyperlink Graph 2012. http://webdatacommons.org/hyperlinkgraph/. http://webdatacommons.org/hyperlinkgraph/2012-08/download.html.
[227] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. 2018. Deconstructing RDMA-enabled Distributed Transactions: Hybrid is Better!. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 233–251.
[228] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. 2015. Fast In-Memory Transaction Processing Using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. 87–104.
[229] Tim S Woodall, Galen M Shipman, George Bosilca, Richard L Graham, and Arthur B Maccabe. 2006. High performance RDMA protocols in HPC. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 76–85.
[230] Zonghan Wu et al. 2020. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems* (2020).
[231] Carl Yang, Aydın Buluç, and John D Owens. 2022. GraphBLAST: A high-performance linear algebra-based graph framework on the GPU. *ACM Transactions on Mathematical Software (TOMS)* 48, 1 (2022), 1–51.
[232] Liang Yao, Chengsheng Mao, and Yuan Luo. 2019. KG-BERT: BERT for knowledge graph completion. *arXiv preprint arXiv:1909.03193* (2019).
[233] 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. *Proceedings of the VLDB Endowment* 6, 7 (2013), 517–528. https://doi.org/10.14778/2536349.2536352
[234] Erfan Zamanian, Carsten Binnig, Tim Harris, and Tim Kraska. 2017. The End of a Myth: Distributed Transactions Can Scale. *Proceedings of the VLDB Endowment* 10, 6 (February 2017), 685–696.
[235] Erfan Zamanian, Xiangyao Yu, Michael Stonebraker, and Tim Kraska. 2019. Rethinking Database High Availability with RDMA Networks. *Proceedings of the VLDB Endowment* 12, 11 (July 2019), 1637–1650.
[236] Ziwei Zhang, Peng Cui, and Wenwu Zhu. 2020. Deep learning on graphs: A survey. *IEEE Transactions on Knowledge and Data Engineering* (2020).
[237] Jie Zhou et al. 2020. Graph neural networks: A review of methods and applications. *AI Open* 1 (2020), 57–81.
[238] Xiaowei Zhu et al. 2020. LiveGraph: A Transactional Graph Storage System with Purely Sequential Adjacency List Scans. *VLDB* 13, 7 (2020), 1020–1034. https://doi.org/10.14778/3384345.3384351
[239] Tobias Ziegler, Sumukha Tumkur Vani, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. Designing Distributed Tree-Based Index Structures for Fast RDMA-Capable Networks. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19)*. 741–758.
[240] Lei Zou, M. Tamer Özsu, Lei Chen, Xuchuan Shen, Ruizhe Huang, and Dongyan Zhao. 2014. GStore: A Graph-Based SPARQL Query Engine. *VLDB Journal* 23, 4 (2014), 565–590. https://doi.org/10.1007/s00778-013-0337-7