GraphScope Flex: LEGO-like Graph Computing Stack

Tao He
Shuxian Hu
Longbin Lai
Dongze Li
Neng Li
Xue Li
Lexiao Liu
Alibaba Group
Hangzhou, China

Xiaojian Luo
Binqing Lyu
Ke Meng
Sijie Shen
Li Su
Lei Wang
Jingbo Xu
Alibaba Group
graphscope@alibaba-inc.com

Wenyuan Yu
Weibin Zeng
Lei Zhang
Siyuan Zhang
Jingren Zhou
Xiaoli Zhou
Diwen Zhu
Alibaba Group
Hangzhou, China

ABSTRACT

Graph computing has become increasingly crucial in processing large-scale graph data, with numerous systems developed for this purpose. Two years ago, we introduced GraphScope as a system addressing a wide array of graph computing needs, including graph traversal, analytics, and learning in one system. Since its inception, GraphScope has achieved significant technological advancements and gained widespread adoption across various industries. However, one key lesson from this journey has been understanding the limitations of a *"one-size-fits-all"* approach, especially when dealing with the diversity of programming interfaces, applications, and data storage formats in graph computing. In response to these challenges, we present GraphScope Flex, the next iteration of GraphScope. GraphScope Flex is designed to be both resource-efficient and cost-effective, while also providing flexibility and user-friendliness through its LEGO-like modularity. This paper explores the architectural innovations and fundamental design principles of GraphScope Flex, all of which are direct outcomes of the lessons learned during our ongoing development process. We validate the adaptability and efficiency of GraphScope Flex with extensive evaluations on synthetic and real-world datasets. The results show that GraphScope Flex achieves 2.4× throughput and up to 55.7× speedup over other systems on the LDBC Social Network and Graphalytics benchmarks, respectively. Furthermore, GraphScope Flex accomplishes up to a 2,400× performance gain in real-world applications, demonstrating its proficiency across a wide range of graph computing scenarios with increased effectiveness.

CCS CONCEPTS

• **Computer systems organization** → *n-tier architectures.*

KEYWORDS

graph computing, distributed system, graph analytics, graph query

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

SIGMOD-Companion '24, June 9–15, 2024, Santiago, AA, Chile

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-0422-2/24/06…$15.00
https://doi.org/10.1145/3626246.3653383

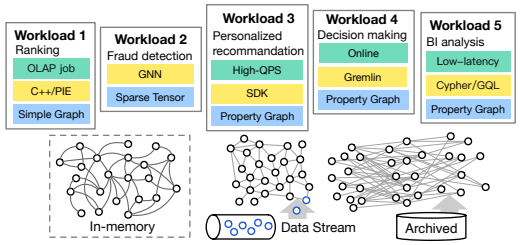


Figure 1: Diversified graph workloads in e-commerce.

ACM Reference Format:
Tao He, Shuxian Hu, Longbin Lai, Dongze Li, Neng Li, Xue Li, Lexiao Liu, Xiaojian Luo, Binqing Lyu, Ke Meng, Sijie Shen, Li Su, Lei Wang, Jingbo Xu, Wenyuan Yu, Weibin Zeng, Lei Zhang, Siyuan Zhang, Jingren Zhou, Xiaoli Zhou, and Diwen Zhu. 2024. GraphScope Flex: LEGO-like Graph Computing Stack. In *Companion of the 2024 International Conference on Management of Data (SIGMOD-Companion '24), June 9–15, 2024, Santiago, AA, Chile.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3626246.3653383

1 INTRODUCTION

Graphs are increasingly becoming the backbone of numerous real-world applications, permeating diverse fields such as social networks, e-commerce, bioinformatics, fintech, and knowledge base [31, 51, 72, 81, 86]. These applications often involve complex interactions and relationships, uniquely represented through graph structures. As the scale of graph data and the intricacy of applications grow, the demand for specialized graph processing systems has escalated. Systems like Pregel [56], Spark GraphX [9], TuGraph [27], TigerGraph [11] and PyG [23] have been developed to address specific graph computation requirements. However, developers often find themselves juggling multiple systems with vastly different programming models and runtime environments. This multiplicity gives rise to a host of issues, such as managing the complexities of data representation, resource scheduling, and performance tuning across these disparate systems.

In response to these challenges, GraphScope [40] was developed as a pioneering solution, designed to offer a comprehensive approach to graph computing. Unlike its predecessors, GraphScope supports a wide range of computing paradigms, enabling it to handle diverse tasks like graph analytics, graph learning, and interactive queries within a single, unified system. Internally, GraphScope employs an extension of Gremlin and a unified dataflow

engine, capable of processing various types of graph computations efficiently. Furthermore, it enhances interoperability with other frameworks in the PyData ecosystem. Despite its ambition to be an efficient, user-friendly, and all-encompassing solution for graph computing, GraphScope has encountered practical challenges.

Real-world example. Figure 1 offers a simplified portrayal of the diversity inherent in graph computing workloads as seen in real-world settings with Alibaba as an example. Here, data is represented as graphs. Within these graphs, vertices symbolize entities such as buyers, sellers, and items. In contrast, edges represent relationships or activities such as buying, selling, or reviewing. These graphs could be either dynamic or static, large or small, and either fit in-memory or exist as extensive historical archives.

These diverse graphs inhabit different graph workloads tuned to cater to a variety of business scenarios. For instance, a ranking workload stipulates the order in which items or sellers appear in search results (marked as Workload 1 in Figure 1). On the other hand, to unravel suspicious entities potentially participating in fraudulent activities to manipulate their ratings and rankings, an anti-fraud task deploying link prediction (SEAL [82]) algorithm is activated (Workload 2). The platform also offers a personalized recommendation service predicting user interests based on historical behaviors and tastes (Workload 3), while a BI analysis-enabled WebUI equips data scientists with the tools necessary to make data-driven decisions (Workload 4), such as identifying the most popular sellers or items in specific regions. Furthermore, some ad-hoc BI workloads sporadically operate over large historical graph data archives (Workload 5), often too large for a graph database's memory due to their infrequent use and high costs.

This example – to some extent mirroring the design motif behind GraphScope – further reasserts the reality that real-world graph computing workloads are diverse, yet they share certain characteristics. Workload 1 encapsulates these shared traits via its utilization of an algorithm akin to PageRank — a category of graph analytics usually demanding high data-intensity and being memory-bound. On the other hand, Workload 2 involves a Graph Neural Network (GNN) task, which often necessitates memory-bound data sampling and CPU/GPU resource-limited computation-intensive back-end training. In contrast, Workloads 4 and 5 are frequently interactive complex query-based graph exploration tasks which typically target a smaller selective subset of the graph based on certain conditions.

Even though GraphScope offers support for distributed immutable in-memory graph storage suitable for static graph processing, real-life workloads often demand varying formats and access paths to the data. As illustrated in Figure 1, visible in the blue section, iterative analytics like k-core and label propagation generally operate over attributed or simple graphs. Conversely, GNN models work on sparse tensors, while interactive queries use a labeled property graph model to facilitate complex attribute-oriented queries. Furthermore, data lifecycle stages also differ — data mining and GNN training usually occur on a snapshot of graph data, while GNN inference and interactive queries function on dynamic graphs receiving continuous updates. BI analysis can occur either over dynamic graphs or historical graph data archives.

While the unified interface proposition from GraphScope might seem beneficial, in practice its one-size-fits-all strategy is ineffective due to the disparate needs of various workloads and user preferences. For instance, data scientists performing interactive queries often prefer domain-specific languages such as Gremlin or Cypher, while developers working on GNN models lean towards specialty libraries such as PyG. Similarly, services necessitating high query throughput might use parameterized queries structured as stored procedures and for graph analytics, iterative algorithms are typically coded in C++ or Java and made accessible as built-in libraries.

Our experience with GraphScope has underscored the limitations of a "one-size-fits-all" approach in the complex arena of graph computing. While requirements can *range widely*, they often *exhibit underlying commonalities*. We've found that a design choice effective for one scenario may fall short in another. No single solution — whether concerning the engine, interface, or storage — can accommodate all varied requirements comprehensively. This divergence between theory and practice revealed through our work on GraphScope, which aimed for a unified solution, led to frustrating trade-offs and imposed certain shortcomings onto specific tasks.

GraphScope Flex. In this paper, we introduce GraphScope Flex, the next iteration of GraphScope. Adopting a modular architecture, GraphScope Flex aims to minimize resource and cost overhead while enhancing deployment flexibility and user experience. Unlike GraphScope's unified dataflow engine, GraphScope Flex disaggregates the engines for various graph tasks, as well as their interfaces and storages. This modular design makes the system architecture akin to a set of LEGO bricks: users can selectively deploy components of GraphScope Flex to streamline the deployment process and adapt the system to their specific needs.

In summary, we make the following contributions:

- We explore graph models, storage schemes, and computing workloads, along with programming interfaces and system performance in graph computing. We also discuss opportunities to enhance composability by decoupling common components. (§2).
- We offer a bird-eye view of the architecture of GraphScope Flex and an in-depth discussion on its modules and techniques.(§3).
- We delve into GRIN for decoupling graph storage from workloads, and key graph storage techniques, such as in-memory Vineyard, dynamic graph store GART and archive format GraphAr(§4).
- We demonstrate GraphScope Flex's flexibility in handling graph queries by using an intermediate representation that supports both Gremlin and Cypher. It executes queries on suitable engines based on their OLAP or OLTP characteristics (§5).
- We examine how GraphScope Flex excels in managing graph analytics by offering a range of built-in algorithms and programming interfaces, supported by both CPU and GPU backends (§6).
- We examine the suitability of GraphScope Flex in catering to diverse requirements in GNN workloads, highlighting its decoupling of sampling and training for independent scaling (§7).
- We present case studies illustrating GraphScope Flex's practical application, efficacy, and versatility in real-world scenarios (§8).
- Lastly, we empirically verify GraphScope Flex's performance. Our findings reveal that it achieves 2.4× the throughput and up to 55.7× speedup on the LDBC Social Network and Graphalytics benchmarks, respectively. Furthermore, it excels in real-world applications, achieving up to 2,400× performance speedup compared to previous solutions (§9).

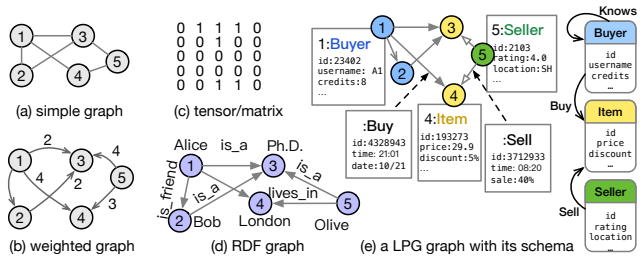


Figure 2: Graph models.

2 BACKGROUND & RELATED WORK

In this section, we present the background on graph data models, storage formats, applications, programming interfaces, performance requirements, and existing graph computing systems. We also review the diversity within these components and highlight opportunities for a disaggregated system design.

2.1 Graph Models & Organizations

Graphs are ubiquitous in real life, effectively modeling complex systems and relationships through vertices and edges. However, the superficial simplicity of graph abstraction conceals a landscape filled with diversity and fragmented nuances. Upon closer inspection, the concept of a graph becomes more complex and diverse than the relational data processing landscape.

Graphs can adopt various data models such as simple graphs, weighted graphs common in graph analytics tasks (like PageRank or SSSP) [1, 9, 56, 83], sparse matrix/tensor used in GNN models [23, 25, 85], RDF observed in knowledge bases [4, 21, 38], or Labeled Property Graph (LPG) [29, 30, 32] widely used in graph databases [7, 11, 20, 27]. Each of these models supports varying operations and provides different access interfaces. Correspondingly, the storage and processing needs also differ substantially, thereby highlighting the need for customizable and flexible solutions in graph computing.

The plurality of graph data models when combined with the myriad ways of graph organization amplifies the diversity, making the landscape of graph processing far from standardized.

2.2 Applications of Graph Computing

Graph computing spans a wide range of applications across domains. We categorize these applications into three main types, each associated with specific graph models and querying paradigms.

Graph Querying. Graph querying involves using specialized languages such as Gremlin [64], Cypher [42], GQL [18], and SPARQL [2] to interrogate and manipulate graph structures. This category primarily includes operations related to pattern matching and complex query formulations. Gremlin and Cypher are often employed for traversal and pattern matching in labeled property graphs, facilitating intricate queries and analyses. On the other hand, SPARQL is predominantly used with RDF graphs, focusing more on sophisticated pattern matching, data aggregation, and integration across various data sources. This distinction highlights the varied nature of graph querying, necessitating flexible and adaptable query processing capabilities in graph computing systems.

Graph Analytics. Graph analytics investigates global structure of graphs using algorithms for clustering, centrality, shortest paths, and reachability. Applications in this category typically utilize simple or weighted graphs. For instance, clustering algorithms like Louvain [34] and centrality measures such as PageRank [60] are central to understanding network dynamics and influence patterns in domains like social network analysis and epidemiology [47].

Graph Learning. Graph learning, especially through Graph Neural Networks (GNNs), applies machine learning techniques to graph-structured data, often represented as sparse tensors or matrices [85]. This process typically involves three key steps: sampling, training, and inference. Sampling is crucial for large graphs to create manageable subsets for efficient processing. During training, models learn from the graph's topology and node features, preparing for tasks like node classification or link prediction. Inference then applies these models to new or evolving data, adaptable to both small-scale and large-scale graph scenarios. Graph learning is vital in fields where relational data patterns play a key role, such as in social network analysis [46, 55, 78] and bioinformatics [79, 84].

2.3 Programming Interfaces

The diversity in graph computing also extends to programming interfaces, each tailored to specific domains within graph operations.

For graph querying, Gremlin [64] and Cypher [42] are widely used. Gremlin, part of Apache TinkerPop [5], offers an extensive set of operators, providing rich expressiveness for graph traversal. However, its robustness comes with complexity, as it includes over 200 steps, many with overlapping functionalities. For instance, steps like valueMap and elementMap both return vertex/edge properties, but with nuanced differences. This complexity poses challenges in ensuring comprehensive support within interactive graph engines. Cypher, initiated by Neo4j [6], has gained wide adoption and significantly contributed to the development of GQL [8], the emerging standard for querying graph databases. The increasing demand for Cypher integration into various systems, including GraphScope, alongside the standardization of ISO/GQL [18], highlights the evolving nature of graph querying interfaces. Additionally, many graph databases offer the capability to register custom stored procedures for enhanced querying functionality.

In graph analytics, following the fixed-point computation, the Pregel API [56] represents a "think-like-a-vertex" interface [57], focusing on vertex-centric computations. PIE [41] allows for handling a partition of a graph as a primary element, offering an alternative methodology. On the other hand, FLASH [53] supports a flexible control flow beyond fixed-point for a wider range of algorithms. In addition, the block-centric model [70], edge-centric model [65], GAS [45] and GraphBLAS [36] are also tailored models for representing graph analytics computation.

Graph learning, especially in training and inference phases, typically employs Python-based interfaces, due to Python's prevalence in the machine learning community. This choice facilitates the integration of graph learning tasks with existing Python-based data science and machine learning ecosystems.

2.4 Performance Requirements

Performance requirements in real-world graph applications exhibit significant diversity, often reflecting the varied nature of the tasks.

Even within a single domain like graph querying, performance expectations can range from high query throughput (QPS) for handling multiple concurrent requests to data parallelism for complex queries typical in BI scenarios. In online services, the emphasis often lies on high availability. Conversely, some applications might prioritize rapid processing of individual tasks, focusing on low latency and efficient single-task execution.

Graph analytics, often aligned with batch processing, also present a spectrum of performance needs. While some can be efficiently conducted in memory, others, especially those involving larger graphs with fewer machines, necessitate out-of-core processing.

Graph learning, particularly in the context of training processes, is predominantly batch-oriented. However, when dealing with large graphs, this often involves a combination of sampling and training to manage the computational demands. This process requires not just efficient data processing, but also a careful balancing of resource utilization to ensure optimal learning outcomes.

These varied performance requirements often coexist and intersect in practical scenarios, underscoring the need for diverse runtime engines and system architectures. Such diversity echoes the principle that a "one-size-fits-all" approach is inadequate in the realm of graph computing.

2.5 Existing Graph Computing Systems

The complex landscape of graph computing, characterized by its diversity in storage abstractions, domain-specific applications, interfaces, and performance requirements, challenges the practicality of a "one-size-fits-all" solution like the original design of GraphScope.

In this varied context, questions arise: Can we, for instance, have a system that enables high-QPS Gremlin querying on a static in-memory graph? Or set up an MPP-like Cypher process on a dynamic graph, while also accommodating GNN training on the same graph? Among state-of-the-art graph computing systems, specialized systems have been developed to address tailored requirements for certain types of graph workloads. For instance, various graph database systems and graph query engines [11, 20, 48, 49, 63, 67, 71] are developed to tailor high-throughput or low-latency query execution with different underlying organization and storage of graphs. Whereas for graph analytic workloads, a different set of siloed systems [9, 33, 35, 41, 58, 61, 74, 76, 87] are designed to excel in parallelizing large-scale analytical computations. To facilitate sampling-based mini-batch training, GNN systems [43, 54, 68, 77] usually have their own graph engines and utilize in-memory storage to maximize the sampling throughputs.

2.6 Opportunity: LEGO-like Modularity

One possible solution to this complexity does not lie in creating siloed, specialized and fragmented systems from scratch for each unique requirement. Instead, the key is in embracing a LEGO-like modularity. This approach involves designing graph computing components – such as various graph storages, runtime engines, workloads, and interfaces – in a way that they can interlock or "plug" into each other seamlessly, much like LEGO bricks.

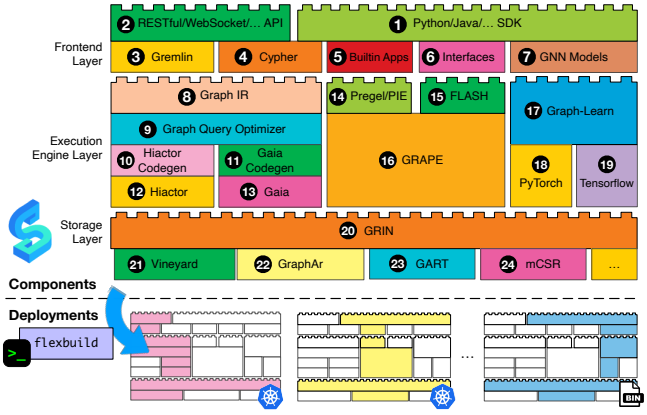


Figure 3: System architecture

In essence, this modular design, which also finds some attempts [62, 73] in other domains, not only addresses the broad spectrum of graph computing requirements but also fosters innovation and adaptability in system architecture, paving the way for more efficient and effective solutions in the field of graph computing.

3 SYSTEM OVERVIEW

Building on the identified opportunity, we introduce GraphScope Flex, a comprehensive system for large-scale graph processing that employs a disaggregated design. It features a modular architecture that reduces resource requirements while providing a seamless, user-friendly experience for flexible deployment. As depicted in Figure 3, GraphScope Flex comprises multiple components, akin to LEGO building bricks. Each component is designed to provide specific functionalities, yet some share underlying commonalities. This modular approach allows users to select components that best align with their specific requirements and build a tailored graph computing stack for their own scenarios. The components are classified into three layers:

Frontend Layer. Graph processing typically necessitates diverse interfaces for distinct tasks. GraphScope Flex offers a range of user interface options, including SDKs (labeled by ①), Web Sockets, and RESTful APIs ②. For graph queries, GraphScope Flex accommodates multiple query languages, specifically Gremlin ③ and Cypher ④. For tasks involving graph analytics and learning, GraphScope Flex features an extensive built-in library containing various common algorithms across different domains, including iterative algorithms ⑤ and GNN models ⑦. To further enhance the functionality of GraphScope Flex, it also provides interfaces ⑥ for the development of new algorithms.

Execution Engine Layer. The components in the execution engine layer are categorized into three specialized groups: The interactive engines for graph querying and pattern matching, the analytical engine for graph analysis, and the learning engine dedicated to graph-based machine learning. A key attribute shared among these engines is their proficiency in efficient, distributed processing of queries and algorithms on large-scale graph data. As a query or algorithm is received, GraphScope Flex compiles it into a distributed execution plan, which is partitioned across multiple compute nodes for parallel processing. Each partition independently operates on

its own compute node and synchronizes with other partitions via a coordinator. This section introduces these engines, with more comprehensive details to be provided in Sections 5 - 7.

Graph Query Engines. Upon receiving a query from the frontend layer, the query is parsed into a unified intermediate representation (GraphIR ⑧). This is followed by optimization through a universal Query Optimizer ⑨ and catalog module. The optimized logical plan employs code generation modules (⑩ and ⑪) to produce the corresponding physical plan. Two execution engines are available, each targeting specific optimization goals. HiActor ⑫, a high-concurrency engine based on the actor model, is optimized for high throughput. In contrast, Gaia ⑬, a dataflow-based engine, focuses on reducing query latency by leveraging data parallelism.

Analytical Engine. The analytical engine accommodates a variety of programming models. This includes the widely adopted vertex-centric model Pregel [56], the PIE model ⑭ based on subgraph-centric programming [41], and the FLASH model ⑮ that supports non-neighbor communications [53]. Underpinning these diverse models is GRAPE ⑯, a distributed high-performance analytical engine. The GRAPE engine provides a set of highly optimized core operators for fragment management, local evaluations on fragments, and their communication. Moreover, it features auto-parallelization of sequential algorithms for distributed environments, as well as GPU acceleration capabilities.

Learning Engine. The learning engine is designated for training GNN models. To facilitate this, GraphLearn ⑰ first samples the graph data and extracts features. Subsequently, these features are organized into batches and dispatched to a backend training engine, which can be either PyTorch ⑱ or TensorFlow ⑲.

Storage Layer. To address the challenges posed by diverse storage formats and data access patterns, GraphScope Flex defines a unified interface ⑳ for graph data management and access. This interface enables seamless integration with various storage backends (㉑-㉔) and makes backend complexities transparent to the execution engines. In Section 4, we will provide further details.

flexbuild and Customized Deployments. Similar to LEGO bricks, not all components are designed to fit together arbitrarily. Within each layer of GraphScope Flex, while components share many core functionalities, each maintains its unique specializations. To help users get a specialized stack for their own needs, we introduce flexbuild, a utility tool that enables users to choose specific components, build and generate their respective binaries or Docker images. These artifacts can be deployed either on a cluster or a single machine, allowing for customized deployments of GraphScope Flex. For instance, in the real-world example described in Section 1, the engineers focusing on Workload 2 might select components ①⑤⑭⑯⑳㉒. Utilizing flexbuild, they can build these components into a Docker image and deploy it on a cluster to provide a service for anti-fraud tasks. In contrast, a data scientist addressing Workload 5 may opt for components ②④⑧④⑨⑪⑬⑳㉓. With flexbuild, these components can be compiled into a binary and run on a single machine for BI analysis. For ease of use, flexbuild simplifies common graph computing tasks with predefined templates and a command-line wizard that assists users in selecting compatible components and generating deployment artifacts.

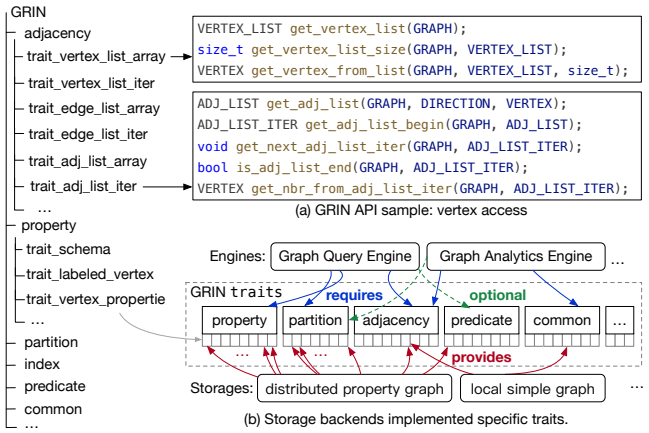


Figure 4: The design of GRIN

4 STORAGE LAYER

The storage layer consists of a set of storage backends in GraphScope Flex, as well as GRIN, a Graph Retrieval INterface above them to provide unified retrieval abilities for the execution layer.

4.1 GRIN, Unified Graph Retrieval Interface

The GRIN in GraphScope Flex is a language-agnostic interface designed to facilitate integration between diverse execution engines and storage backends. It provides comprehensive and well-defined APIs for graph retrieval, tailored to accommodate the varied graph retrieval requirements of execution engines and the distinct data models and access patterns of storage backends. This design not only simplifies the implementation of retrieval functionalities across different systems but also ensures that storage backends can clearly communicate their capabilities and limitations. The essence of GRIN lies in its ability to enable various storage backends to "interlock" effectively with engines, embodying the modular, LEGO-like approach central to the GraphScope Flex architecture.

In particular, the GRIN is defined in C language, which makes it portable to systems written in other languages. Based on a thorough analysis of the graph retrieval requirements of the execution engines in GraphScope Flex, it abstracts the requirements into six categories as shown in the left side of Figure 4. The `adjacency` category forms the core of graph abstractions, such as vertices and edges. Based on it, the `property` and `partition` categories cover the data models of property graphs and partitioned graphs, which are widely used in graph databases and distributed graph processing systems respectively. The remaining three categories of `index`, `predicate` and `common` are designed to address the auxiliary graph operations and common system requirements, such as indexing, predicate-pushdown and error handling.

To further characterize the graph retrieval requirements, handles (*e.g.*, `ADJ_LIST`) and APIs (*e.g.*, `get_adj_list(...)`) for graph entities and their operations are defined under several `traits` within each category. For example, the array-like and iterator-based access are two common ways to traverse a list structure, and the `adjacency` category contains corresponding `traits` for list handles, such as vertex and adjacent list. Figure 4(a) illustrates the APIs defined under the array-like access `trait` for vertex list and the iterator-based access `trait` for adjacent list.

For storage backends, they can only provide the `traits` that are feasible for their own system capabilities and limitations. Similarly, an engine will may only require or optionally support some `traits`. In this way, a graph analytical algorithm such as PageRank will work on a property graph, as property graph storages will provide the `traits` for `adjacency` and `partition` as shown in Figure 4(b).

4.2 Storage Backends

As shown in Figure 3, GraphScope Flex offers various storage backends implemented GRIN APIs. In this subsection, we will introduce some of them, each tailored to specific use cases.

Vineyard. Vineyard [80] is an in-memory immutable data manager that offers out-of-the-box high-level abstraction and zero-copy sharing for distributed data in big data tasks, such as graph analytics, numerical computing and machine learning.

In GraphScope Flex, Vineyard serves as the backend storage for in-memory graphs. It adopts the property graph data model and handles graph partitioning using edge-cut partitioning. In order to optimize graph retrieval, Vineyard provides various built-in indices such as CSR and CSC representations for graph structures, and internal ID assignment to vertices. These features allow Vineyard to effectively implement most of the GRIN `traits`.

GART. Graph data is not always static. Sometimes both the graph topology and the properties of vertices/edges may be updated. To accommodate such scenarios, GraphScope Flex has incorporated a mutable in-memory graph storage, GART [66], which supports multi-version concurrency control (MVCC) for dynamic graph data.

Specifically, GART always provides consistent snapshots of graph data (identified by a *version*), and it updates the graph with the version number *write_version*. For read operations, a compact graph representation like CSR is the optimal choice, though it suffers from the costly overheads of write operations. Conversely, adjacency lists based on linked lists are efficient for write operations but perform poorly in read operations due to inadequate data locality. To ensure high performance for both read and write operations, GART employs an efficient and mutable CSR-like data structure.

GraphAr. GraphAr [17] (short for "Graph Archive") is a standardized file format designed for efficient storage of graph data on both local and cloud file systems. It is developed on top of Apache ORC [13] and Parquet [14], two widely used columnar storage formats in the big data ecosystem. GraphAr serves as the default persistent format for GraphScope Flex, improving the performance of data loading and graph construction. Additionally, it can be directly used as a data source for applications by integrating GRIN.

One of the key features of GraphAr is its ability to efficiently partition graph data into multiple data chunks, allowing for targeted retrieval and parallel processing without the burden of loading the full graph into memory. Furthermore, GraphAr empowers certain graph-related operations to be executed directly at the storage layer, such as retrieving vertices with a specific label or fetching the neighbors of a given vertex, using its built-in indexes. These capabilities improve data management and performance, making GraphAr an effective archive format for interoperability in graph computing. In GraphScope Flex, the lack of overlapping storage backends removes the need for replica maintenance, while GraphAr serves as the intermediary for infrequent data exchanges between them.

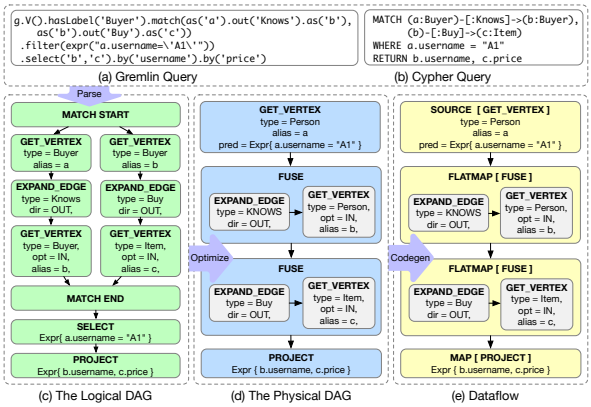


Figure 5: Compilation process of IR-based interactive stack.

5 GRAPH QUERYING

Our work in GraphScope[40] introduced an interactive engine employing the Gremlin[64] language for queries, complemented by the Gaia [63] engine for their distributed execution. However, as applications burgeoned on GraphScope, we confronted multifaceted challenges pertaining to both the query interface and the runtime.

As mentioned in Section 2.3, users may have varying preferences for query languages, such as Gremlin and Cypher. Additionally, the upcoming standardization of ISO/GQL [18] also necessitates the development of a new query interface.

On the runtime front, Gaia's design, being a data-parallel, batching system, is tailored for parallel execution of fairly intricate queries on large graphs - a natural fit for the OLAP domain. Yet many other applications fit the OLTP paradigm, demanding efficient processing of small, high-concurrency queries. Gaia, due to its design, struggles to fulfill these specific OLTP requirements.

To address these challenges, we have developed a novel interactive stack of GraphScope Flex for graph queries. This stack integrates a graph *Intermediate Representation* (IR) abstraction, designed to capture the shared functionalities across diverse query interfaces. Accompanying this is an optimizer anchored in equivalent transformation rules specific to IR, along with two specialized code generators for the Gaia [63] and HiActor [52] engines. These engines adeptly manage OLAP and OLTP workloads, respectively.

5.1 The IR Abstraction

The IR abstraction aims to encapsulate common functionalities present across various graph query interfaces. This approach ensures that the parser, optimizer, and code generators are developed in an unified manner, circumventing duplicate efforts. As an illustration, consider Gremlin and Cypher shown in Figure 5, both of which express the query: "finding the purchased items' prices of friends." While the two queries evidently differ in syntax (they might also diverge in query semantics [28], though exploring this further lies outside the scope of this paper), they exhibit *common* functionalities that blend graph operations with relational operations. A case in point is their shared execution of graph pattern matching to identify items purchased by friends, followed by relational projection to extract specific properties of interest during the

search process. The evolving ISO/GQL query language [18] also demonstrates similar characteristics.

To address this, we devised the IR abstraction, which is tailored to semantically encapsulate the fundamental commonalities inherent in various graph query languages. At its core, the IR abstraction defines a data model D and a set of operators Ω. The model D manifests as a schema-like structure, where each data field has a String-typed name complemented by a specific data type. Such data types can range from primitive (like Integer, Float, String) to composite (such as List, Map), or most importantly graph-associated types. The graph-associated types encompass Vertex, Edge, and Path, each carrying supplementary properties. For example, a *Buyer* vertex v, as depicted in Figure 2, has *id* that is a unique identifier, *type* that is *Buyer*, and properties of *username* and *credits*.

Each operator in Ω processes a set of data tuples from D and yields another set of data tuples. These operators can be divided into two categories: graph operators and relational operators.

- **Graph Operators:** Central to Ω are operators specific to graph operations. They include actions such as expanding adjacent edges from vertices (`EXPAND_EDGE`), retrieving end vertices from edges (`GET_VERTEX`), and executing pattern matching (`MATCH_START`, `MATCH_END`). These operators are illustrated in Figure 5(c).
- **Relational Operators:** This category emphasizes operations that project property values from vertices and edges (`PROJECT`), filter vertices and edges based on specific criteria (`SELECT`), and structure results by sequence (`ORDER`) or by categorization (`GROUP`). Such operators are commonly used with relational databases.

For any pair of operators o_1 and o_2 within Ω, a data connection can be established between them if the output data tuples from o_1 align with the input requirements of o_2. This capability is crucial for constructing a directed acyclic graph (DAG) that encapsulates the computational logic of a query, often referred to as the computational DAG. This DAG will be used in two distinct stages:

- **Logical Stage:** This is the initial stage where the query is parsed and transformed into a semantic representation. It primarily focuses on capturing the semantic of the query, independent of how the query will be executed.
- **Physical Stage:** Derived from the logical stage through optimization processes, the physical stage concretizes the execution plan for the query. It delineates the specific operations and execution order required to fulfill the query.

Figure 5(c) illustrates the logical DAG, the semantic interpretation of both the Gremlin and Cypher queries. Next, we will delve into the physical DAGs and their role in query execution.

5.2 IR-based Optimizer

Given the logical DAG, the fundamental goal of the IR-based optimizer is to convert it into an optimized physical DAG for efficient execution. This transformation is achieved through a combination of Rule-Based Optimization (RBO) and Cost-Based Optimization (CBO), taking into account both semantic equivalence and the physical context, such as the capabilities of the underlying graph store, which can be deduced from the GRIN interface (refer to Section4.1).

Before diving into the details, we define a pattern graph as a concise graph p, and the process of matching this pattern involves identifying all subgraphs within the data graph G that are isomorphic to p. For instance, both the Gremlin and Cypher queries in Figure 5 involve matching a pattern graph that navigates from a *Buyer* vertex "a" to an *Item* vertex "c" via another *Buyer* vertex "b". One of the matched instance in the data graph of Figure 2 can be written as *{a:1, b:2, c:3}*, where each key-value pair, such as *a:1*, indicates mapping from a pattern vertex "a" to a data vertex "1".

Rule-based Optimization. The RBO process involves the application of a set of predefined heuristic rules to the logical DAG, ensuring that the query's semantics remain unaltered. We highlight two rules that are most commonly applied:

EdgeVertexFusion. In numerous graph queries, the `EXPAND_EDGE` operator is frequently succeeded by a `GET_VERTEX` operator, indicating the retrieval of neighboring vertices rather than adjacent edges. To streamline this process, we apply the **EdgeVertexFusion** rule, which merges the `EXPAND_EDGE` and `GET_VERTEX` operators into a unified, fused operator whenever it is pragmatically possible. The conditions under which these two operators can be combined vary. For example, fusion may not be feasible in distributed scenarios where property retrieval is required in `GET_VERTEX`. An illustration of this fused operator after applying the rule is shown in Figure 5(d).

FilterPushIntoMatch. Both the `GET_VERTEX` and `EXPAND_EDGE` operators in our design are capable of accepting a predicate as a parameter, enabling immediate filtering of vertices and edges upon retrieval from the graph store. While it is a common practice for users to apply the `SELECT` operation after pattern matching, as exemplified by the queries in Figure 5, our system implements the **FilterPushIntoMatch** rule to optimize this process. This rule actively pushes the predicate from the `SELECT` operator into the pertinent graph operators. The application of the **FilterPushIntoMatch** rule serves a dual purpose: it not only diminishes the volume of intermediate data, enhancing performance, but also facilitates the possible downward propagation of predicates, optimizing data retrieval at the store level. As illustrated in Figure 5(d), the predicate `a.username = "A1"` is effectively pushed into the `GET_VERTEX` operator associated with vertex "a", exemplifying this optimization in action.

Cost-based Optimization. We have incorporated insights from our previous research, GLogue [49], to process CBO for the Interactive stack. Given that graph pattern matching is a crucial and computationally intensive component [28] of graph queries, optimizing its execution is the main focus of GLogue. In GLogue, our approach entails tracking patterns ranging from the smallest, single-vertex patterns to the largest, encompassing patterns with up to k vertices, along with an estimation of each pattern's frequency. Here, the term "frequency" of p refers to the count of matched instances in G. As the execution plan for matching a pattern graph p inevitably requires the computation of various subgraphs (a subgraph is defined as a graph comprising a subset of the original graph's vertices or edges), the cost of an execution plan can be determined by summing the estimated frequencies of all relevant subgraphs, retrievable from GLogue. Consequently, the execution plan with the lowest associated cost is considered optimal.

In Figure 5, the transition from the logical DAG to the physical DAG, orchestrated by the cost-based optimization (CBO) process, results in an obvious structural transformation. Initially, the logical DAG exhibits a bifurcated structure, which is then altered into a

linear chain in the physical DAG. This restructuring is particularly evident in the treatment of the vertices aliased as "b". By merging these vertices in the physical DAG, we effectively eliminate the need for a separate scanning operation for the "b"-aliased vertex, thereby reducing the associated cost.

5.3 Code Generation

The final step of compilation transforms the non-executable physical Directed Acyclic Graph (DAG) into executable code, varying for the Gaia engine in Online Analytical Processing (OLAP) or the Hiactor engine in Online Transaction Processing (OLTP). The Gaia engine, using a dataflow model similar to our Intermediate Representation (IR), processes data with operators like `MAP` and `FLATMAP`. We map each IR operator in the DAG to a corresponding Gaia operator, maintaining connections between operators. This transformation is depicted in Figure 5(e). Initially, the `GET_VERTEX` operator is converted into a `SOURCE` operator, initiating the computation. Then, the operators intended for neighbor retrieval are mapped to `FLATMAP` operators, representing the one-to-many vertex-to-neighbor relationship. Lastly, the `PROJECT` operator is adapted into a `MAP` operator, transforming each vertex into its property value.

6 GRAPH ANALYTICS

For graph analytics, GraphScope Flex provides user-friendly interfaces and a high-performance graph analytical engine GRAPE [41].

User-Friendly Interfaces. For the convenience of users, GraphScope Flexoffers built-in algorithm packages. These packages feature APIs that are compatible with NetworkX [3], GraphX [9], and Giraph [1] interfaces, enabling users to enjoy the performance improvements offered by the software without having to modify the original code implemented in other systems.

If the built-in packages cannot meet user needs, the software provides Python/C++/Java SDKs with various programming paradigms, allowing users to employ the programming paradigm with which they are most familiar to implement complex algorithmic logic. For example, users can use the Pregel API to implement vertex-centric algorithms, or they can write sequential algorithms using the subgraph-centric PIE model [41]. Alternatively, users can utilize the FLASH model [53], which supports algorithms that utilize non-neighbor communication and offers great expressive capability.

High Performance Analytical Engine. At the core of the analytics stack in GraphScope Flex is GRAPE [41], a distributed graph computing engine with the capability for auto-parallelization of sequential algorithms. To enhance its capabilities, we have incorporated Ingress [44] to facilitate algorithm auto-incrementalization, supplementing the generality of GRAPE's PIE model. GRAPE supports two types of backend to execute the graph analytic algorithms:

The CPU backend. GRAPE supports acceleration of graph operators with SIMD components such as AVX2 and AVX512 on the CPU backend, and emphasizes on optimizing communication and memory overhead across multiple nodes. In terms of communication, GRAPE trades latency for throughput. It aggregates fragmented, randomly distributed small messages in memory into a continuous compact buffer before dispatching them all at once, thus enhancing bandwidth utilization. Furthermore, it employs *varint* encoding and perfect hash to reduce peak memory usage.

The GPU backend. Since many graph algorithms can significantly benefit from GPU acceleration, we have integrated GPU support into the GRAPE engine. While GPU cores are often less powerful individually than CPU cores, they compensate with a greater number of cores and higher memory bandwidth. For intra-GPU, GRAPE integrated multiple load balance thread mapping and GPU-friendly data structure [59] to improve the GPU utilization. For inter-GPU, GRAPE employs a work-stealing strategy to dynamically balance GPU workloads [58]. Idle GPU cores will steal work from busy ones to maximize GPU utilization on the fly.

7 GRAPH LEARNING

Optimizing distributed GNN training on large-scale graphs in industrial settings is challenging. First, there's an imbalance in computational demands between sampling and training. GPU training often has much higher throughput than CPU-based sampling, leading to poor resource utilization and suboptimal overall throughput. Second, the network and I/O overheads from distributed graph sampling, feature extraction, and data transfer between host and device significantly extend training sample preparation time. The learning stack in GraphScope Flex addresses these issues by effectively parallelizing and pipelining the sampling and training processes.

Decoupled Sampling & Training. Given the observation that the computational demands of sampling and training are asymmetrical, the learning stack adopts a decoupled design that physically isolates the sampling and training processes. This design supports independent scaling of sampling and training to accommodate optimal resource utilization and enhance the training throughput. For instance, due to the high costs of GPU instances, it is often more economically efficient to deploy a CPU cluster for graph sampling and feature collection, while reserving GPU instances for training.

Asynchronous Pipelining. To overlap sampling computation and network communication, sampling processes are designed to concurrently perform subgraph sampling and feature collection for multiple batches. The process of multi-hop graph sampling is modeled as a dataflow, where each node represents a sampling hop, and the edge indicate the data dependency between consecutive hops. Feature collection is listed as the sink node in the dataflow. Since graphs are partitioned in a distributed manner, each node in the dataflow is parallelized across the graph partitions as distinct tasks. While awaiting the completion of a specific task, the sampling process schedules the execution of other tasks (belonging to the same or different batches) to avoid being blocked by a single task. To prevent training processes from idling while waiting for inputs, a prefetch mechanism is employed to continuously retrieve data from the sample channel to a prefetch cache for each training process.

Compatible with Open-source GNN Stack The learning stack supports both TensorFlow [25] and PyTorch [22] as the training backend. To enrich the model library that users can train with GraphScope Flex, the data-layer APIs in learning stack are designed to be compatible with PyTorch Geometric (PyG) [23]. PyG models can be trained in GraphScope Flex with minimal modifications.

8 USE CASES

To check the flexibility and effectiveness of the modular and disaggregated design of GraphScope Flex, we have deployed it in a

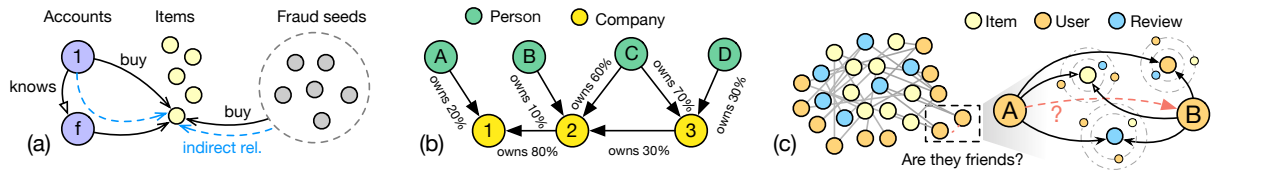


Figure 6: Use cases

variety of real-world scenarios. Below, we delve into some of these instances to showcase the diverse deployments of GraphScope Flex.

Real-time Fraud Detection. E-commerce platforms process tens of thousands of customer orders every second. Among these, some may be fraudulent. For instance, certain orders might forge genuine purchasing behavior to artificially boost the popularity of specific items. As a simplified example of real-time detection, it is to identify suspicious transactions by checking each order against *fraud seeds* – accounts previously identified with known frauds. For an account with `id=1`, the following Cypher query checks for direct or indirect co-purchasing with *fraud seeds*, as depicted in Figure 6(a):

```
MATCH (v:Account{id:1})-[b1:BUY]->(:Item)<-[b2:BUY]-(s:Account)
WHERE s.id IN SEEDS AND b1.date-b2.date < 5 /*within 5 days*/
       WITH v, COUNT(s) AS cnt1
MATCH (v)-[:KNOWS]-(f:Account), (f)-[b1:BUY]->(:Item)<-[b2:BUY]-(s:
      Account) WHERE s.id IN SEEDS WITH v, cnt1, COUNT(s) AS cnt2
WHERE w1 * cnt1 + w2 * cnt2 > threshold
RETURN v
```

To facilitate such queries, we deploy GraphScope Flex for OLTP graph queries, utilizing HiActor as the computing engine, and GART as the storage engine. When an order is placed, an *(Account)-[Buy]-(Item)* edge is added to GART, leveraging its dynamic graph storage capability. This query is executed in GraphScope Flex. If the query returns any records, indicating that the weighted average of the number of direct and indirect relationships exceeds a predefined `threshold`, an alert will be triggered. This preemptive step is vital to prevent direct lodging of potentially fraudulent orders.

Equity Analysis. In the context of financial analysis, analysts frequently aim to identify the dominant shareholders responsible for steering a company, typically those who cumulatively hold more than 51% of the company's shares. It is non-trivial to find the answer in real-world scenarios. For instance, Company 1 in Figure 6(b) is owned by Person A alongside a sequence of corporate shareholders. The objective for analysts is to ascertain the genuine controller of Company 1, *i.e.,* Person C, who holds 51% of the shares: 0.8×0.6 through Company 2 and $0.8 \times 0.3 \times 0.7$ via Company 3.

This task can be tackled using a GraphScope Flex deployment equipped with analytical modules. In graph modeling, both shareholders and companies are represented as vertices, while investments are depicted as edges. Each edge carries a weight, indicating the share percentage. Within the frontend layer, users opt to implement a modified label propagation algorithm with the GraphX API. This algorithm calculates the shares for each company during each propagation iteration. Subsequently, the algorithm is bundled into a *jar* and executed on GRAPE, the high-performance analytical engine in the engine layer of GraphScope Flex. In the storage layer, GRIN selects the in-memory store Vineyard as the storage backend to managing the rarely modified graph data.

Social Relation Prediction. Social relations on e-commerce platforms are crucial, as they provide insights into user behavior and preferences, facilitating targeted marketing and personalized recommendations. However, these platforms often lack a complete social network of users, unlike social network platforms. Graph Neural Networks (GNNs), such as NCN [75], are employed to predict potential unobserved relationships between users, thereby enriching the social connections on e-commerce sites. The NCN's sampling phase, depicted in Figure 6(c), involves extracting first-order common neighbors for each training edge's vertices and performing k-hop subgraph sampling around each common neighbor. This task is particularly challenging in terms of computational demand and complexity, especially for large-scale graphs.

The learning stack in GraphScope Flex can efficiently support training NCN on billion-scale graphs. Depending on the graph scale, the sampling servers in GraphLearn can be flexibly scaled out to enhance the sampling throughput. To accommodate heterogeneous hardware and improve resource utilization in sampling servers, the learning stack also supports configuring the sampling devices (CPU or GPU) and the concurrency of sampling processes in each server. Training servers asynchronously pull the sampling results from sampling servers and can be scaled to match the sampling throughput. As the original social relation graph remains unchanged and will be frequently accessed during training, Vineyard is selected as the storage backend due to its I/O efficiency. Graph data in Vineyard are accessed by sampling processes through GRIN.

Cybersecurity Monitoring. Cybersecurity represents a paramount concern for numerous enterprises, with a particular emphasis on thwarting Trojan attacks. We detailed the task and its Gremlin-based solution in our previous work [40]. With flexbuild, users can effortlessly select pertinent components from GraphScope Flex to construct a tailored graph BI stack optimized for this specific task.

9 EVALUATIONS

In this section, we evaluate GraphScope Flex's capability of efficient processing of large graphs for both synthetic workloads and the real-world applications described in Section 8.

9.1 Supported Scenarios

Before delving into the performance evaluation, we initially analyze the capabilities of GraphScope Flex in comparison to GraphScope. By incorporating diverse components with specialized functionalities, GraphScope Flex significantly expands the range of supported scenarios in graph computing compared to its predecessor. Table 1 outlines the scenarios supported by each system.

9.2 Synthetic Workloads

Using synthetic workloads, we assessed the performance of the storage layer and various deployments on GraphScope Flex. The datasets and their abbreviations are listed in Table 2. If not otherwise mentioned, all experiments were conducted on a managed K8s

Table 1: Supported scenarios comparison.

Scenario	GraphScope	GS Flex
Python and Gremlin interface	✓	✓
Querying graph in Cypher	×	✓
Complex offline graph queries	✓	✓
High-throughput online graph queries	×	✓
Iterative graph analytics	✓	✓
Sampling and training on graph data	✓	✓
Computing on an in-memory graph	✓	✓
Computing on a graph with persistent store	×	✓
Graph view synced with a relational DB	×	✓
Graph data sharing with other graph systems	×	✓

cluster consisting of 8 nodes. Each node was equipped with dual 26-core Intel(R) Xeon(R) Platinum CPUs at 2.50GHz and 768 GB of memory. The nodes were interconnected via a 50 Gbps network.

Exp-1. Storage Performance. We evaluate the effectiveness of GRIN and test the performance of GART and GraphAr, which are the newly introduced storage backends in GraphScope Flex.

Firstly, to showcase effectiveness of GRIN which makes backend complexities transparent to the execution engines, we conduct experiments on three applications with varying storage backends.

The applications are PageRank(on `CF`), BI-Querying(on `SNB-30`) and GNN-Training(on `PD`) which are typical workloads of the graph analytics, interactive query and graph learning respectively, while the backends are Vineyard, GART, and GraphAr introduced in Section 4.2. Using GRIN APIs to access the graph data, each application is implemented only once and can be deployed on different storage backends. We report the execution time of PageRank, average querying time of BI queries and one-batch training time of GNN in Figure 7(a). We can see that all the combinations can generate correct results in a reasonable time. Generally, Vineyard is the fastest backend due to its in-memory and immutable design. GART is slower, as its more complex architecture accommodates dynamic updates. GraphAr, being the slowest, incurs extra I/O overheads for direct data retrieval due to its archiving-focused design.

To further demonstrate the effectiveness of GRIN, we compare the performance of GraphScope Flex with the baseline (*i.e.,* the original GraphScope without GRIN). In GraphScope, the execution engines are tightly coupled with the only default storage backend Vineyard, so that the graph retrieval implementations are specifically optimized for Vineyard and cannot be easily extended to support other backends. As shown in Figure 7(b), one may find that the performance of GraphScope Flex with GRIN is comparable to baseline for all three applications, with only a slight overhead less than 8%. This means that GRIN does not introduce significant overheads to GraphScope Flex.

Next, we demonstrate the efficiency of GART, which provides high performance read operations on graph data while allowing updates to graph data. To see this, we compared the read performance (measured by edge scan throughput) with a state-of-the-art dynamic graph storage LiveGraph [88] and a *static* graph storage CSR, on four datasets (`UK`, `CF`, `TW` and `SNB-30`). Note that the performance of CSR is the upper bound of a dynamic graph storage as it assumes the graph is immutable. As shown in Figure 7(c), on average GART outperforms LiveGraph 3.88× on read performance, and can achieve 73.5% throughput compared with static graph storage.

Table 2: Datasets used in synthetic workloads.

Abbr.	Dataset	|V|	|E|
`FB0`	datagen-9_0-fb [19]	12.8M	1.05B
`FB1`	datagen-9_1-fb [19]	16.1M	1.34B
`ZF`	datagen-9_2-zf [19]	434.9M	1.04B
`G500`	graph500-26 [19]	32M	1.05B
`WB`	webbase-2001 [37]	118M	1.71B
`UK`	uk-2005 [37]	39.5M	1.57B
`CF`	com-friendster [50]	65.6.5M	1.81B
`TW`	twitter-2010 [50]	41.7M	1.47B
`IT`	it-2004 [37]	41M	1.15B
`AR`	arabic-2005 [37]	22.7M	1.11B
`PD`	ogbn-products [26]	2.4M	62M
`PA`	ogbn-papers100M [26]	111M	1.6B
`SNB-30`	Datasets (-x for scale factor x) generated for LDBC social network benchmark [10]	89M	541M
`SNB-300`		817M	5.27B
`SNB-1000`		2.69B	17.79B

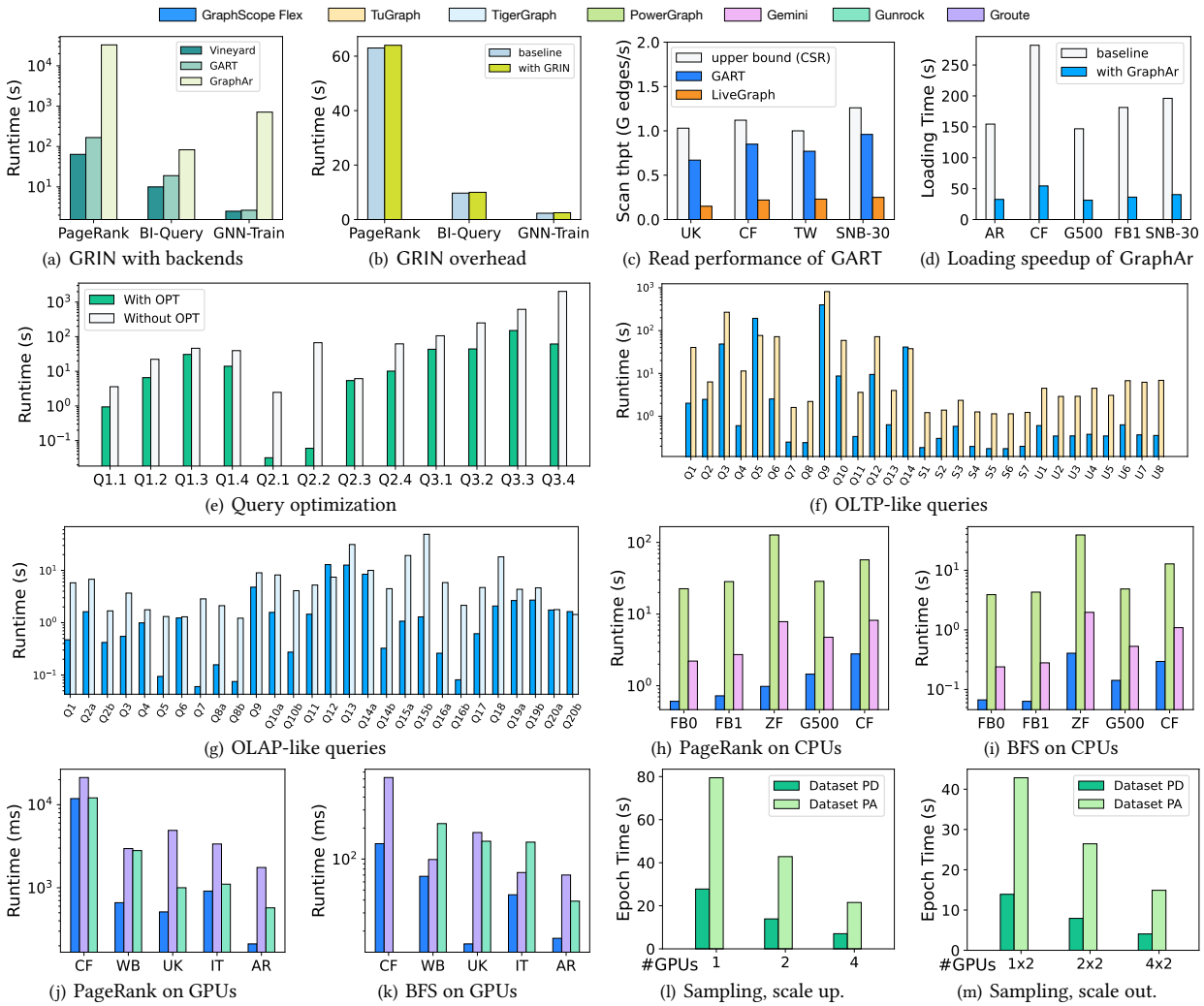
Lastly, we compare the performance of building graphs from external storages in the format of GraphAr with the baseline, where the datasets are in CSV format. The results are shown in Figure 7(d). We can see that GraphAr can significantly improve the performance of graph construction, with a speedup of around 5× for all datasets.

Exp-2. Graph Query Performance. We evaluate the performance of GraphScope Flex for processing graph queries, demonstrating the effectiveness of the query optimization including RBO and CBO (Section 5), and the efficiency of the deployments of GraphScope Flex for handing OLTP and OLAP queries, respectively.

To evaluate the effectiveness of query optimization in GraphScope Flex, based on the LDBC Social Network Business Intelligence (SNB-BI) workloads [69], we generated three distinct sets of queries, which are referred to as Q1, Q2, and Q3 as given in [24]. Each set comprises four queries designed to specifically test different optimization strategies: the *EdgeVertexFusion* and *FilterPushIntoMatch* rules in RBO, and the CBO strategy. The experiments were conducted on a single node of the cluster. Figure 7(e) shows the performance gain with RBO optimization: the *EdgeVertexFusion* rule yielded an average speedup of 2.9×, and the *FilterPushIntoMatch* rule achieved an impressive average speedup of 279×. Similarly, queries optimized with CBO performed 11× better compared to those without CBO.

For the performance of GraphScope Flex in processing OLTP-like queries, we present the official audit results from the LDBC Social Network Interactive Benchmark [39]. We deployed GraphScope Flex as the OLTP graph querying stack, with HiActor serving as the computing engine. To align with the existing reports in [15], the benchmark was conducted on a single machine with a CPU of 24 cores and 384GB of memory. Figure 7(f) reports the auditing results [15] of GraphScope Flex against TuGraph on the `SNB-300` dataset, displaying the average latency for 14 complex queries (C1-C14), 7 short queries (S1-S7), and 8 update queries (U1-U8) in the benchmark. The results show GraphScope Flex outperforms TuGraph in all queries except C5, achieving an average speedup of 8.92×. Furthermore, the recorded throughput for GraphScope Flex is 33,261 ops/s, which is 2.45× higher than TuGraph's 13,532 ops/s.

To assess the performance of GraphScope Flex in handling OLAP-like queries, we utilized the LDBC SNB-BI workloads [69]. In this



(a) GRIN with backends (b) GRIN overhead (c) Read performance of GART (d) Loading speedup of GraphAr

(e) Query optimization (f) OLTP-like queries

(g) OLAP-like queries (h) PageRank on CPUs (i) BFS on CPUs

(j) PageRank on GPUs (k) BFS on GPUs (l) Sampling, scale up. (m) Sampling, scale out.

Figure 7: Performance evaluation

experiment, we configured GraphScope Flex with Gaia as the distributed computing engine, as outlined in Section 5 that is tailored for OLAP queries. We conducted the SNB-BI benchmark evaluation on the `SNB-1000` dataset of the first snapshot that encompasses over 99% of the total data, executing a total of 20 queries (some queries having different variants) with the frequency and parameters specifying by official auditing. As comparison, we gathered auditing results from TigerGraph [16], the state-of-the-art performance of the SNB-BI benchmark. Our tests were conducted on a cluster of 4 machines that matched TigerGraph's settings, including 64 CPU cores (128 threads), and a total memory of 1.5TB. The average latency for all queries are displayed in Figure 7(g). With the exception of query 12 and 20b, GraphScope Flex outperforms TigerGraph by an average speedup of 10×. This is mainly due to GraphScope Flex's sophisticated optimizations and flexible deployment for various scenarios.

Exp-3. Graph Analytics Performance. To test the analytics performance, we conducted evaluations using the LDBC Graphalytics Benchmarks [19], comparing GraphScope Flex with the state-of-the-art CPU-based and GPU-based graph processing systems [12]. We report the performance of PageRank and breadth-first search (BFS) algorithms here, *w.l.o.g.* Figures 7(h) to 7(k) display the performance of these algorithms across different systems and datasets, with GraphScope Flex consistently outperforming others. Compared with the CPU-based systems, it is on average 54.6× (resp. 4.2×) faster than PowerGraph (resp. Gemini), up to 129.6× (resp. 8.0×). For GPU-based systems, it is on average 3.3× (resp. 3.3×) faster than Groute (resp. Gunrock), up to 9.5× (resp. 9.9×). For all dataset used, GraphScope Flex outperforms other CPU-based and GPU-based graph processing systems, and can process large graphs that other systems failed to process. This is primarily attributed to GraphScope Flex's adaptive optimization, selecting optimal computing kernels for various graph data, and topology-aware communication that efficiently distributes workloads across GPU cluster units [58], accounting for asymmetries like NVLink to maximize high-bandwidth link usage.

Exp-4. Graph Learning Performance. We evaluate the scalability of the learning stack in GraphScope Flex by training a 3-layer GraphSAGE model on the `PA` and `PD` datasets. The sampling fan-out is set to [15, 10, 5] and the batch size is 1024. The nodes allocated in this experiment are additionally equipped with 4 Nvidia A10 GPUs. In both the single-node and distributed experiments, GPU sampling

Table 3: Throughput of the real-time fraud detection.

#threads	10	20	30	40
Throughput	98,907	184,826	279,005	355,813

is used with the number of sampling processes set equal to the number of GPUs. Figure 7(l) shows the results of scaling up experiments. By increasing the number of GPUs, the end-to-end training time per epoch decreases linearly. This is because each sampling process handles a subset of the sampling workloads independently in single-node training, increasing the number of sampling processes can improve the training efficiency accordingly. The results of scaling out experiments are presented in Figure 7(m). The number of used GPUs in each node is fixed at 2. By increasing the number of nodes from 1 to 4, we can observe an almost-linear boost in training performance. This result shows that, despite longer batch execution times in sampling processes due to network communication costs in distributed sampling and feature collection, the asynchronous pipelining and prefetch mechanism ensure nearly linear scaling out performance in the learning stack of GraphScope Flex .

9.3 Real-World Applications

GraphScope Flex has been widely deployed in production at Alibaba, and it supports about 50,000 graph jobs every day. Next, we report results from production for the applications described in Section 8.

Exp-5. Real-time Fraud Detection. Our evaluation of the real-time fraud detection was conducted on an extensive segment of a real-life transaction graph. Our assessment went beyond the Cypher query discussed in Section 8, incorporating several queries that explore various relationships. In this scenario, numerous clients continuously placed orders, each of which triggered a set of mandatory queries, reflecting diverse relational checks. These queries were executed across different CPU thread configurations in GraphScope Flex, showcasing a range of cloud computational resources from 10 to 40 threads. The results, detailed in Table 3, demonstrate the system's scalable nature, with throughput almost linearly increasing with the number of active working threads. Compared to the limitations of previous deployments, which struggled under peak load, GraphScope Flex demonstrates a more than 30× improvement in throughput under similar configurations. This highlights GraphScope Flex's aptitude for meeting the dynamic and high-volume requirements of real-world applications.

Exp-6. Equity Analysis. We evaluated the performance of equity analysis using a graph constructed from open data on the equity relationships of registered companies in China. This graph includes 0.3 billion vertices and over 1.5 billion edges. We compared the GraphScope Flex deployment, as detailed in Section 8, against an existing SQL-based baseline. The baseline, which stored data in relational tables, checked each tuple (*i.e.,* a company) and calculated the shares among its shareholders. Despite incorporating numerous approximations to reduce computational costs, the baseline was unable to produce complete results. It allowed only a limited number of tuples to be involved in the queries and required about 1 hour to process a small subset of the data. In contrast, GraphScope Flex could generate all results within 15 minutes on the whole graph, enabling its daily running in production and ensuring that analysts always have access to the latest results.

Exp-7. Social Relation Prediction. The performance of prediction was evaluated by training an NCN model on an in-house social relation dataset, which consists of 10 million vertices and 200 million edges. The training job was executed on a cluster comprising 30 nodes. In order to optimize end-to-end throughput, tailored to the demands of sampling and training workloads, 10 nodes were designated for sampling while the remaining 20 nodes were utilized for training. The end-to-end training time per epoch is 1.5 hours, which can be linearly scaled for large-scale graphs in production.

Exp-8. Cybersecurity Monitoring. With GraphScope Flex, we upgrade the application reported in [40]. Practice shows that GraphScope Flex can subsume GraphScope without any performance degradation, offering a concise and easy-to-maintain deployment instead. Using graph traversal written in Gremlin, GraphScope Flex achieves a speedup of 2,400× over the equivalent SQL queries. Since the Trojan detection queries are two-hop graph traversals, they avoid the costly join operations required in SQL queries.

Summary. We find the following. (1) The composition and functionality of the components within GraphScope Flex are both effective and efficient. (a) GRIN effectively supports various storage backends with a maximum performance degradation of 8%. (b) GraphScope Flex outperforms leading systems in LDBC Social Network Benchmarks, achieving 2.45× higher throughput than TuGraph in SNB Interactive workloads, and an average of 10× smaller latency than TigerGraph in SNB-BI workloads, respectively. (c) In graph analytics, GraphScope Flex exceeds both CPU-based (Gemini, Powergraph) and GPU-based (Groute, Gunrock) systems, reaching up to 55.7 × (*resp.* 9.9×) faster performance. (d) GraphScope Flex shows strong scalability in learning tasks, with up to 3.94 × (*resp.* 3.42×) improvement when scaling from $1(1 \times 2)$ GPUs to $4(4 \times 2)$ GPUs. (2) Better still, GraphScope Flex performs well in real-world applications. (a) It manages 355,813 qps in real-time fraud detection, and significantly outperforming SQL-based solutions in cybersecurity by over 2,400×. (b) It facilitates comprehensive full-data analysis in equity analysis. and (c) It efficiently enables the daily training of social relation prediction models and offers linear scalability.

10 CONCLUSION AND FUTURE WORK

GraphScope Flex addresses the limitations of the *"one-size-fits-all"* solution by adopting a modular architecture, and users can selectively deploy components of GraphScope Flex to meet their specific requirements. Evaluations on both synthetic and real-life cases show that GraphScope Flex can efficiently and flexibly handle diverse application scenarios. Moving forward, GraphScope Flex will persist in its evolution towards a high-performance and user-friendly computing system for large-scale graph processing. Future efforts will encompass these directions: (1) Enhancing existing components and introducing new ones, such as GQL support in the frontend layer and additional storage backends in the storage layer. (2) Developing a graph-specific ETL or transformation language to streamline the integration of applications across different graph models derived from the same dataset. (3) Additionally, in scenarios blending graph tasks with SQL-like operations, a unified compiler across multiple engines can significantly enhance workflow interoperability and expand the scope of graph computations.

REFERENCES

[1] 2011. Apache Giraph. https://giraph.apache.org.
[2] 2013. W3C Sparql 1.1 Query Language. https://www.w3.org/TR/sparql11-query/.
[3] 2014. NetworkX. https://networkx.org/.
[4] 2014. W3C, Resource Description Framework (RDF). https://www.w3.org/RDF/.
[5] 2015. Apache TinkerPop. https://tinkerpop.apache.org.
[6] 2015. Cypher Query Language in Neo4j. https://neo4j.com/product/cypher-graph-query-language/.
[7] 2017. JanusGraph. https://janusgraph.org/.
[8] 2018. GQL Standard. https://https://www.gqlstandards.org.
[9] 2018. Spark GraphX. https://spark.apache.org/graphx/.
[10] 2019. LDBC Social Network Benchmark. https://ldbcouncil.org/benchmarks/snb/.
[11] 2019. TigerGraph. https://www.tigergraph.com/.
[12] 2022. Performance report of LDBC Graphalytics. https://github.com/alibaba/libgrape-lite/blob/master/Performance.md.
[13] 2023. Apache ORC. https://orc.apache.org.
[14] 2023. Apache Parquet. https://parquet.apache.org/.
[15] 2023. Auditing results of LDBC SNB Interactive Workload. https://ldbcouncil.org/benchmarks/snb-interactive/.
[16] 2023. Full Disclosure Report for TigerGraph of the LDBC Social Network Benchmark. https://ldbcouncil.org/benchmarks/snb/LDBC_SNB_BI_20221109_SF1000_tigergraph.pdf.
[17] 2023. GraphAr. https://github.com/alibaba/GraphAr.
[18] 2023. ISO graph query standard GQL. https://www.gqlstandards.org/.
[19] 2023. LDBC Graphalytics. https://ldbcouncil.org/benchmarks/graphalytics/.
[20] 2023. Neo4j. https://neo4j.com/.
[21] 2023. OntoText. https://www.ontotext.com/.
[22] 2023. PyTorch. https://github.com/pytorch/pytorch.
[23] 2023. PyTorch Geometric. https://github.com/pyg-team/pytorch_geometric.
[24] 2023. Queries used for the experiment of graph query optimization. https://github.com/alibaba/GraphScope/tree/main/flex/resources/queries/examples/store_procedure.
[25] 2023. TensorFlow. https://github.com/tensorflow/tensorflow.
[26] 2023. The Open Graph Benchmark (OGB). https://ogb.stanford.edu/.
[27] 2023. TuGraph, The Distributed Graph Database Behind AliPay. https://tugraph.antgroup.com/.
[28] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. 2017. Foundations of Modern Query Languages for Graph Databases. *ACM Comput. Surv.* 50, 5, Article 68 (sep 2017), 40 pages.
[29] Renzo Angles, Angela Bonifati, Stefania Dumbrava, George Fletcher, Alastair Green, Jan Hidders, Bei Li, Leonid Libkin, Victor Marsault, Wim Martens, et al. 2023. PG-Schema: Schemas for property graphs. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–25.
[30] Dmitry Anikin, Oleg Borisenko, and Yaroslav Nedumov. 2019. Labeled property graphs: SQL or NoSQL?. In *2019 Ivannikov Memorial Workshop (IVMEM)*. IEEE, 7–13.
[31] Saeid Azadifar, Mehrdad Rostami, Kamal Berahmand, Parham Moradi, and Mourad Oussalah. 2022. Graph-based relevancy-redundancy gene selection method for cancer diagnosis. *Computers in Biology and Medicine* 147 (2022), 105766.
[32] Nico Baken. 2020. Linked data for smart homes: Comparing RDF and labeled property graphs. In *LDAC2020—8th Linked Data in Architecture and Construction Workshop*. 23–36.
[33] Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. 2017. Groute: An Asynchronous Multi-GPU Programming Model for Irregular Computations. In *ACM Sigplan Symposium on Principles and Practice of Parallel Programming*. 235–248.
[34] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. 2008. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment* 2008, 10 (2008), P10008.
[35] Rong Chen, Jiaxin Shi, Yanzhe Chen, Binyu Zang, Haibing Guan, and Haibo Chen. 2019. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. *ACM Transactions on Parallel Computing (TOPC)* 5, 3 (2019), 1–39.
[36] Timothy A. Davis. 2019. Algorithm 1000: SuiteSparse:GraphBLAS: Graph Algorithms in the Language of Sparse Linear Algebra. *ACM Trans. Math. Softw.* 45, 4 (2019).
[37] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (dec 2011), 25 pages. https://doi.org/10.1145/2049662.2049663
[38] Stefan Decker, Sergey Melnik, Frank Van Harmelen, Dieter Fensel, Michel Klein, Jeen Broekstra, Michael Erdmann, and Ian Horrocks. 2000. The semantic web: The roles of XML and RDF. *IEEE Internet computing* 4, 5 (2000), 63–73.
[39] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. 2015. The LDBC Social Network Benchmark: Interactive Workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 619–630.
[40] Wenfei Fan, Tao He, Longbin Lai, Xue Li, Yong Li, Zhao Li, Zhengping Qian, Chao Tian, Lei Wang, Jingbo Xu, et al. 2021. GraphScope: a unified engine for big graph processing. *Proceedings of the VLDB Endowment* 14, 12 (2021), 2879–2892.
[41] Wenfei Fan, Wenyuan Yu, Jingbo Xu, Jingren Zhou, Xiaojian Luo, Qiang Yin, Ping Lu, Yang Cao, and Ruiqi Xu. 2018. Parallelizing sequential graph computations. *ACM Transactions on Database Systems (TODS)* 43, 4 (2018), 1–39.
[42] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An evolving query language for property graphs. In *Proceedings of the SIGMOD 2018*. 1433–1445.
[43] Swapnil Gandhi and Anand Padmanabha Iyer. 2021. P3: Distributed deep graph learning at scale. In *15th {USENIX} OSDI 21*. 551–568.
[44] Shufeng Gong, Chao Tian, Qiang Yin, Wenyuan Yu, Yanfeng Zhang, Liang Geng, Song Yu, Ge Yu, and Jingren Zhou. 2021. Automating Incremental Graph Processing with Flexible Memoization. *Proc. VLDB Endow.* 14, 9 (may 2021), 1613–1625.
[45] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs.. In *OSDI*, Vol. 12. 2.
[46] Zhiwei Guo and Heng Wang. 2020. A deep graph neural network-based mechanism for social recommendations. *IEEE Transactions on Industrial Informatics* 17, 4 (2020), 2776–2783.
[47] Petter Holme. 2017. Three faces of node importance in network epidemiology: Exact results for small graphs. *Physical Review E* 96, 6 (2017), 062305.
[48] Kasra Jamshidi, Rakesh Mahadasa, and Keval Vora. 2020. Peregrine: a pattern-aware graph mining system. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.
[49] Longbin Lai, Yufan Yang, Zhibin Wang, Yuxuan Liu, Haotian Ma, Sijie Shen, Bingqing Lyu, Xiaoli Zhou, Wenyuan Yu, Zhengping Qian, Chen Tian, Sheng Zhong, Yeh-Ching Chung, and Jingren Zhou. 2023. GLogS: Interactive Graph Pattern Matching Query At Large Scale. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, Boston, MA, 53–69.
[50] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.
[51] Michelle M Li, Kexin Huang, and Marinka Zitnik. 2022. Graph representation learning in biomedicine and healthcare. *Nature Biomedical Engineering* 6, 12 (2022), 1353–1369.
[52] Su Li et al. 2023. Hiactor: an open-source hierarchical actor framework. https://github.com/alibaba/hiactor.
[53] Xue Li, Ke Meng, Lu Qin, Longbin Lai, Wenyuan Yu, Zhengping Qian, Xuemin Lin, and Jingren Zhou. 2023. FLASH: A Framework for Programming Distributed Graph Processing Algorithms. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 232–244.
[54] Tianfeng Liu, Yangrui Chen, Dan Li, Chuan Wu, Yibo Zhu, Jun He, Yanghua Peng, Hongzheng Chen, Hongzhi Chen, and Chuanxiong Guo. 2023. {BGL}:{GPU-Efficient}{GNN} Training by Optimizing Graph Data {I/O} and Preprocessing. In *20th USENIX Symposium on NSDI 23*. 103–118.
[55] Zhiwei Liu, Liangwei Yang, Ziwei Fan, Hao Peng, and Philip S Yu. 2022. Federated social recommendation with graph neural network. *ACM Transactions on Intelligent Systems and Technology (TIST)* 13, 4 (2022), 1–24.
[56] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 135–146.
[57] Robert Ryan McCune, Tim Weninger, and Greg Madey. 2015. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys (CSUR)* 48, 2 (2015), 1–39.
[58] Ke Meng, Liang Geng, Xue Li, Qian Tao, Wenyuan Yu, and Jingren Zhou. 2023. Efficient Multi-GPU Graph Processing with Remote Work Stealing. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. 191–204.
[59] Ke Meng, Jiajia Li, Guangming Tan, and Ninghui Sun. 2019. A Pattern Based Algorithmic Autotuner for Graph Processing on GPUs. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming* (Washington, District of Columbia) *(PPoPP '19)*. ACM, New York, NY, USA, 201–213.
[60] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1998. *The pagerank citation ranking: Bring order to the web.* Technical Report. Technical report, stanford University.
[61] Santosh Pandey, Lingda Li, Adolfy Hoisie, Xiaoye S Li, and Hang Liu. 2020. C-SAW: A framework for graph sampling and random walk on GPUs. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.
[62] Pedro Pedreira, Orri Erling, Konstantinos Karanasos, Scott Schneider, Wes McKinney, Satya R Valluri, Mohamed Zait, and Jacques Nadeau. 2023. The composable data management system manifesto. *Proceedings of the VLDB Endowment* 16, 10 (2023), 2679–2685.
[63] 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)*. USENIX Association, 321–335.

[64] Marko A Rodriguez. 2015. The gremlin graph traversal machine and language (invited talk). In *Proceedings of the 15th Symposium on Database Programming Languages*. 1–10.

[65] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 472–488.

[66] Sijie Shen, Zihang Yao, Lin Shi, Lei Wang, Longbin Lai, Qian Tao, Li Su, Rong Chen, Wenyuan Yu, Haibo Chen, Binyu Zang, and Jingren Zhou. 2023. Bridging the Gap between Relational OLTP and Graph-based OLAP. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, 181–196.

[67] Li Su, Xiaoming Qin, Zichao Zhang, Rui Yang, Le Xu, Indranil Gupta, Wenyuan Yu, Kai Zeng, and Jingren Zhou. 2022. Banyan: A Scoped Dataflow Engine for Graph Query Service. *Proc. VLDB Endow.* 15, 10 (2022), 2045–2057.

[68] Jie Sun, Li Su, Zuocheng Shi, Wenting Shen, Zeke Wang, Lei Wang, Jie Zhang, Yong Li, Wenyuan Yu, Jingren Zhou, and Fei Wu. 2023. Legion: Automatically Pushing the Envelope of Multi-GPU System for Billion-Scale GNN Training. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, Boston, MA, 165–179.

[69] Gábor Szárnyas, Jack Waudby, Benjamin A. Steer, Dávid Szakállas, Altan Birler, Mingxi Wu, Yuchen Zhang, and Peter Boncz. 2022. The LDBC Social Network Benchmark: Business Intelligence Workload. *Proc. VLDB Endow.* 16, 4 (2022), 877–890.

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

[71] Vasileios Trigonakis, Jean-Pierre Lozi, Tomáš Faltín, Nicholas P Roth, Iraklis Psaroudakis, Arnaud Delamare, Vlad Haprian, Călin Iorgulescu, Petr Koupy, Jinsoo Lee, et al. 2021. {aDFS}: An Almost {Depth-First-Search} Distributed {Graph-Querying} System. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 209–224.

[72] Ke Tu, Wei Qu, Zhengwei Wu, Zhiqiang Zhang, Zhongyi Liu, Yiming Zhao, Le Wu, Jun Zhou, and Guannan Zhang. 2023. Disentangled Interest importance aware Knowledge Graph Neural Network for Fund Recommendation. In *Proceedings of the 32nd ACM International Conference on Information and Knowledge Management*. 2482–2491.

[73] Suketu Vakharia, Peng Li, Weiran Liu, and Sundaram Narayanan. 2023. Shared foundations: Modernizing meta's data lakehouse. In *The Conference on Innovative Data Systems Research, CIDR*.

[74] Keval Vora. 2019. {LUMOS}:{Dependency-Driven} Disk-based Graph Processing. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 429–442.

[75] Xiyuan Wang, Haotong Yang, and Muhan Zhang. 2023. Neural Common Neighbor with Completion for Link Prediction. arXiv:2302.00890 [cs.LG]

[76] Yangzihao Wang, Yuechao Pan, Andrew Davidson, Yuduo Wu, Carl Yang, Leyuan Wang, Muhammad Osama, Chenshan Yuan, Weitang Liu, Andy T Riffel, et al. 2017. Gunrock: GPU graph analytics. *ACM Transactions on Parallel Computing (TOPC)* 4, 1 (2017), 3.

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

[78] Liangwei Yang, Zhiwei Liu, Yingtong Dou, Jing Ma, and Philip S Yu. 2021. Consisrec: Enhancing gnn for social recommendation via consistent neighbor aggregation. In *Proceedings of the 44th international ACM SIGIR conference on Research and development in information retrieval*. 2141–2145.

[79] Hai-Cheng Yi, Zhu-Hong You, De-Shuang Huang, and Chee Keong Kwoh. 2022. Graph representation learning in bioinformatics: trends, methods and applications. *Briefings in Bioinformatics* 23, 1 (2022), bbab340.

[80] Wenyuan Yu, Tao He, Lei Wang, Ke Meng, Ye Cao, Diwen Zhu, Sanhong Li, and Jingren Zhou. 2023. Vineyard: Optimizing Data Sharing in Data-Intensive Analytics. *Proc. ACM Manag. Data* 1, 2 (2023).

[81] Mohamad Zamini, Hassan Reza, and Minou Rabiei. 2022. A review of knowledge graph completion. *Information* 13, 8 (2022), 396.

[82] Muhan Zhang and Yixin Chen. 2018. Link Prediction Based on Graph Neural Networks. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems* (Montréal, Canada) *(NIPS'18)*. Curran Associates Inc., Red Hook, NY, USA, 5171–5181.

[83] P Zhang and G Chartrand. 2006. *Introduction to graph theory*. Tata McGraw-Hill.

[84] Xiao-Meng Zhang, Li Liang, Lin Liu, and Ming-Jing Tang. 2021. Graph neural networks and their current applications in bioinformatics. *Frontiers in genetics* 12 (2021), 690049.

[85] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. 2020. Graph neural networks: A review of methods and applications. *AI open* 1 (2020), 57–81.

[86] Zhilun Zhou, Yu Liu, Jingtao Ding, Depeng Jin, and Yong Li. 2023. Hierarchical Knowledge Graph Learning Enabled Socioeconomic Indicator Prediction in Location-Based Social Network *(WWW '23)*. Association for Computing Machinery, New York, NY, USA, 122–132.

[87] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A Computation-Centric Distributed Graph Processing System.. In *OSDI*. 301–316.

[88] Xiaowei Zhu, Guanyu Feng, Marco Serafini, Xiaosong Ma, Jiping Yu, Lei Xie, Ashraf Aboulnaga, and Wenguang Chen. 2019. LiveGraph: A Transactional Graph Storage System with Purely Sequential Adjacency List Scans. *Proceedings of the VLDB Endowment* 13, 7 (2019).